CICS and the JVM server
Developing and Deploying Java applications

- Explore OSGi and the new JVM server
- CICS runtime
- Understand the benefits of OSGi
- Use CICS Explorer SDK to develop and deploy Java applications

Chris Rayns
George Bogner
Tony Fitzgerald
Elisabetta Flamini
Jun Hua Gao
Ivan D Hargreaves
Phil Wakelin
Hongtao Wen
Bei Chun Zhou

ibm.com/redbooks
CICS and the JVM server: Developing and Deploying Java applications

January 2013
Note: Before using this information and the product it supports, read the information in “Notices” on page ix.
Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant 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.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will 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.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.
Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

- AIX®
- CICS Explorer®
- CICSPlex®
- CICS®
- DB2®
- IBM®
- IMS™
- Language Environment®
- OMEGAMON®
- OS/390®
- OS/400®
- Rational®
- Redbooks®
- Redbooks (logo) ®
- System z®
- Tivoli®
- VTAM®
- WebSphere®
- zArchitecture®
- z/OS®
- zSeries®

The following terms are trademarks of other companies:

- Linux is a trademark of Linus Torvalds in the United States, other countries, or both.
- Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.
- Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.
- UNIX is a registered trademark of The Open Group in the United States and other countries.
- Other company, product, or service names may be trademarks or service marks of others.
Preface

This IBM® Redbooks® publication provides information about the new Java virtual machine (JVM) server technology in IBM CICS® Transaction Server for z/OS® V4.2. We begin by outlining the many advantages of its multi-threaded operation over the pooled JVM function of earlier releases. The Open Services Gateway initiative (OSGi) is described and we highlight the benefits OSGi brings to both development and deployment.

Details are then provided about how to configure and use the new JVM server environment. Examples are included of the deployment process, which takes a Java application from the workstation Eclipse integrated development environment (IDE) with the IBM CICS Explorer® software development kit (SDK) plug-in, through the various stages up to execution in a stand-alone CICS region and an IBM CICSPlex® environment.

The book continues with a comparison between traditional CICS programming, and CICS programming from Java. As a result, the main functional areas of the Java class library for CICS (JCICS) application programming interface (API) are extensively reviewed.

Further chapters are provided to demonstrate interaction with structured data such as copybooks, and how to access relational databases by using Java Database Connectivity (JDBC) and Structured Query Language for Java (SQLJ).

Finally, we devote a chapter to the migration of applications from the pooled JVM model to the new JVM server run time.

Authors

This book was produced by a team of specialists from around the world working at the IBM International Technical Support Organization (ITSO), Poughkeepsie Center.

Chris Rayns is the CICS Project Leader and writes extensively on all areas of CICS. He is an IT Specialist at the ITSO, Poughkeepsie Center. Before joining the ITSO, Chris worked in IBM Global Services in the UK as a CICS IT Specialist.
**George Bogner** is a Software IT Specialist working in IBM Sales and Distribution supporting the CICS Transaction Server family product suite. George has worked at IBM for 28 years, specializing in the database/data communication (DB/DC) area working with CICS, IBM IMS™, and IBM DB2® supporting client accounts in IBM Global Services. He currently works out of Raleigh, North Carolina, supporting North American customers by providing them CICS Smart Seminars, proof of technology (POT), proof of concept (POC), and consulting services for CICS related topics.

**Tony Fitzgerald** is a Software Support Specialist in the UK. He has worked for IBM for 13 years as a CICS Level 2 Support specialist supporting clients in the UK and Europe. He has 23 years experience working with CICS and DB2 as a Systems Programmer and as an Applications Programmer. He holds a degree in Computer Science and Management Science from the University of Keele. Tony has contributed to several other Redbooks publications and also ITSO projects sponsored by IBM Learning Services.

**Elisabetta Flamini** is a Software Engineer in CICS Development at Hursley Park, UK. She has a Masters degree in Nuclear Engineering from the University “La Sapienza” of Rome and a certificate in Computer Security from the University of Oxford. She worked for seven years as a Software Engineer and L3 Support Engineer for the IBM Tivoli® Workload Scheduler product, in the Rome Tivoli Lab. She joined CICS Development in 2008. Her areas of expertise on CICS are Java and Web services.

**Jun Hua Gao** is a Staff Software Engineer in China. He has four years of experience in CICS. He holds a Masters degree of Computer Science in JiLin University from China. His areas of expertise include CICS JVM server, WebService CTG, and Event Processing. He has written extensively on JVM server diagnostics.

**Ivan D Hargreaves** is a Software Engineer in CICS Development at Hursley Park. He has 18 years of industry experience, 13 of which have been working in CICS. Ivan holds a first class M.Eng (hons) degree in Computer Systems Engineering (1996), and joined IBM in 2000. Ivan has been Lead Developer/Designer of the JVM server technology since its inception. More recently, he has lead and designed the adoption of IBM WebSphere® Application Server Liberty Technologies in CICS.

**Phil Wakelin** works for IBM UK in Hursley, and is a member of the CICS strategy and planning teaming. He has worked with many different CICS technologies for the last 20 years. He currently has responsibility for new functionality in the areas of CICS interconnectivity, CICS integration, and Java support. He is the author of many white papers, SupportPacs, and IBM Redbooks in the area of CICS Transaction Processing.
Hongtao Wen is a Senior IT Specialist in China. He has more than 15 years of experience in mainframe software. He has led the technical selling support of IBM WebSphere pillar products on IBM System z® (in China) since 2005.

Bei Chun Zhou is a CICS L3 software engineer in the China Development Library. He has two years of experience in CICS. Bei holds a Masters degree in Computer Science (2010). His areas of expertise include CICS, Java, and z/OS. He wrote extensively on Chapter 8 “Interacting with structured data”.

Thanks to the following people for their contributions to this project:

Rufus P. Credle Jr., Tamikia Lee
International Technical Support Organization, Poughkeepsie Center

George Burgess, Paul Cooper, Ben Cox, Ivan D Hargreaves, Mark Pocock, Graham Rawson, David Roberts, John Tilling, Matthew Webster, William Yates
IBM UK Software Group, Hursley

Now you can become a published author, too!

Here’s an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- Use the online Contact us review Redbooks form found at:
  ibm.com/redbooks
Stay connected to IBM Redbooks

- Find us on Facebook:
  http://www.facebook.com/IBMRedbooks
- Follow us on Twitter:
  http://twitter.com/ibmredbooks
- Look for us on LinkedIn:
  http://www.linkedin.com/groups?home=&gid=2130806
- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:
- Stay current on recent Redbooks publications with RSS Feeds:
  http://www.redbooks.ibm.com/rss.html
In Part 1 of this book, we introduce the technologies that contribute to the cutting edge Java support in CICS Transaction Server (TS) for z/OS Version 4.2 (V4.2). In particular, we investigate the new CICS Java virtual machine (JVM) server model, detail its configuration, drive its enablement, and discover how CICS TS V4.2 has introduced the benefits of Open Services Gateway initiative (OSGi) to the JVM server.

The OSGi architecture is described at length, and we see how its implementation through CICS bundles and the CICS Explorer software development kit (SDK) provides everything from a cohesive development environment to a powerful deployment model. With these underlying technologies, Java applications in CICS can be packaged and managed effectively, their entire lifecycle can be controlled, and those applications and their dependencies can by dynamically versioned.
In the latter chapters, our attention focuses more on the deployment process of a Java application from the workstation integrated development environment (IDE), by using Eclipse with the CICS Explorer SDK plug-in. We venture through the various stages up to the execution in a stand-alone CICS region and a CICSPlex environment.

Finally, we dedicate a section to the basic failure scenarios that you might meet during setup and application development. A number of tools and techniques are described that can help you discover and fix problems in your application or configuration.
Chapter 1. Introduction

Java has grown rapidly in popularity throughout the Information Technology (IT) industry and is now the programming language of choice for many organizations. Java has been available as a programming language in CICS since CICS Transaction Server (TS) V1.3. Support has steadily improved over a number of releases. However, the advents of CICS TS V4.2 is of particular significance for modern workloads.

CICS TS V4.2 fully uses the powerful JVM server model, it embraces modern dynamic application development technologies (through Open Services Gateway initiative (OSGi) technology), and it introduced a rich client development platform via the CICS Explorer software development kit (SDK). Further capacity and workload benefits are realized with the move to 64-bit storage for the underlying JVM, through the support for the IBM 64-bit SDK for z/OS, Java Technology Edition, Version 6.0.1 providing significant gains in performance and scalability over previous versions of CICS Java support.

Yet more benefits have been realized by embracing the CICS bundle resource for packaging, deployment, and controlling the application lifecycle. CICS bundle exploitation, combined with the support for dynamic versioning and dependency management brought by OSGi, achieves great advances in the manageability and deployment of Java applications to CICS.
Three main topics are covered in this chapter:

- A brief Java overview
- A history of Java support in CICS
- An overview of the new features in CICS TS V4.2

1.1 Java overview

Java has become a popular programming language and runtime environment for building new applications on all platforms, including IBM System z. In many mainframe installations, new programs are still created in languages such as COBOL, C/C++, or PL/I, but with the increasing use of middleware, such as IBM WebSphere Application Server, the Java programming model has expanded greatly. However, Java is also used in traditional transaction managers, such as CICS TS and IBM IMS, and database managers such as IBM DB2.

Using Java in CICS is the theme of this book and we expand on the exciting new CICS features available to Java programmers throughout the text. However, outside CICS, on System z you can also run Java directly under the z/OS UNIX System Services command line, or from job control language (JCL). Even further afield, CICS provides an Eclipse-based development tool (CICS Explorer SDK) for client-based development of your Java applications.

1.1.1 Java specification

The Java specifications are maintained by the Oracle Corporation, but other companies, IBM in particular, provide input for these specifications. The Java language is based on the philosophy *develop once, run everywhere.*

Java was originally developed in the early 90's by Sun Microsystems Inc. as an object-oriented programming language. The syntax used in Java code originates from C++ and is therefore recognizable for those with C++ experience. However, there are significant differences between C++ and Java:

- The Java programming model was expanded with numerous application programming interfaces (APIs) that are organized in libraries. The Java Platform, Enterprise Edition programming model goes even further and defines a syntax for writing and packaging enterprise applications.
- Java programs are not compiled into persistent load modules. However, technologies were added over the years to do compiles dynamically while running the applications. The first technology to support this was the just-in-time compiler (JIT). Later on, more technologies were included in the Java virtual machine (JVM) to speed up this process.
Java, or actually the JVM, has its own memory management. The programmer does not need to worry about memory allocation and de-allocation because the JVM does this for you.

Over the years, Java grew to be popular, and the JVM is available on nearly all available operating systems, such as Microsoft Windows, Apple OS-X, IBM OS/400®, Linux, UNIX, and z/OS UNIX.

1.1.2 Java virtual machine

At the core of Java is the Java virtual machine (JVM), a complete software microprocessor with its own instruction set and operation (op)-codes. The JVM provides automatic memory management, garbage collection, and other functions for the programmer.

The IBM JVM, and most other JVMs, are implemented through licensed source code from the Oracle Corporation. The source code is provided in C and Java and is highly portable. IBM ported it to many operating systems: IBM AIX®, IBM OS/400, z/OS, and others.

The JVM uses z/OS UNIX System Services for z/OS for base operating system functions. Therefore, you must correctly install and tune UNIX System Services to get optimum performance from the JVM.

The JVM is the essence of Java because it provides the machine independence that is seen as its most significant advantage. Although the JVM is not a unique concept and there were other software microprocessors over the past 20 years, it is the first and only one to achieve broad acceptance.

1.2 History of JVM support in CICS

In the next few sections, we review the history of JVM support in CICS, starting at CICS TS V1.3 and going through to version CICS TS V4.2.

1.2.1 CICS Transaction Server V1.3

CICS TS 1.3 was the first level of CICS Transaction Server to support Java, at the Java Development Kit (JDK) 1.1 level. Initially, however, Java programs were not run under a JVM. Instead, they were compiled using the High Performance Java (HPJ) product, which produces a Java Program Object. HPJ took the Java bytecodes and compiled them to IBM S/390® machine code. The programs then ran natively under the control of IBM Language Environment®, just like a C application. Obviously, with this approach you lose platform independence.
Because the startup cost for an HPJ-compiled application is high, much like that of a JVM, a later program temporary fix (PTF) for CICS TS 1.3 introduced a technique called *hot-pooling*. With hot-pooling enabled, the HPJ environment is saved and reused between applications.

JVM support (for JDK 1.1 only) was introduced with another PTF to CICS TS V1.3. However, the JDK 1.1 JVM was not designed for reuse, a JVM was initialized, used, and then terminated for each CICS JVM program, which made it expensive in terms of path length. Realistically, this single use JVM was only good for programs that were either run infrequently or were long-lived.

At that time, JVM support was only provided in CICS TS V1.3 to provide the ability to run those parts of the Java application programming interface (API) that were not supported by HPJ to provide Java compliance.

### 1.2.2 CICS Transaction Server V2.1 and V2.2

These two releases of CICS supported Java at the *IBM Developer Kit for OS/390®, Java 2 Technology Edition at the SDK 1.3.1 level*. The Java 1.1 JVM supported by CICS TS V1.3 was no longer supported. Any Java programs that ran under CICS TS V1.3 had to be migrated to Java 2 to run under the reusable JVM.

HPJ and hot-pooling (described in 1.2.1, “CICS Transaction Server V1.3” on page 5) were still supported in CICS TS V2.1 and CICS TS V2.2, but only for migration purposes. The stated direction then was that future development should use the JVM.

Support for the resettable JVM was introduced at Version 2. This meant that a JVM could be started and used to run consecutive Java applications, resetting its storage areas between each program invocation. This was the first step towards the goal of having a single JVM serve multiple transactions, which is shown when we describe the JVM server in CICS TS V4.2.

There were restrictions. One restriction for CICS programs running under the JVM, in these two releases, was that a JVM program cannot link, either directly or indirectly, to another JVM program in the same CICS region. This was also true for CICS TS V1.3.

### 1.2.3 CICS Transaction Server V2.3

This level of CICS supported JDK 1.4.1 of the *IBM Software Developer Kit for z/OS, Java 2 Technology Edition*. It provided support for the continuous JVM and introduced the shareable class cache, which allowed for classes and the results of JIT compilation to be shared between several JVMs.
Figure 1-1 shows the various JVM types.

<table>
<thead>
<tr>
<th>Single use JVM (REUSE=NO)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create JVM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Continuous JVM (REUSE=YES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create JVM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resettable JVM (REUSE=RESET) *removed in CICS TS 3.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create JVM</td>
</tr>
</tbody>
</table>

*Figure 1-1  JVM types*

The restriction mentioned in 1.2.2, “CICS Transaction Server V2.1 and V2.2” on page 6, was lifted. It was now possible to have more than one JVM program in the link stack, meaning that one JVM program can link to another JVM program (directly or indirectly).

### 1.2.4 CICS Transaction Server V3.1

The level of the JDK supported by CICS TS V3.1 was the *IBM SDK for z/OS, Java 2 Technology Edition, Version 1.4.2*.

This release of CICS was the first that introduced support for Web services in CICS. These new capabilities extended CICS applications into a service-oriented architecture (SOA) world, enabling them to be exposed directly as Web services.

Another important enhancement at CICS TS V3.1 was the addition of channels and containers. This new feature provided an alternative method of passing data between CICS programs that addressed restrictions found when using commareas. This in turn meant that Java class library for CICS (JCICS) support was enhanced to add a support channels and containers mechanism for inter-program data transfer. See 7.6.4, “Communicating through channels and containers” on page 210 for an example on how to use the new API.
1.2.5 CICS Transaction Server V3.2

CICS TS V3.2 supported two versions of Java:

- IBM SDK for z/OS, Java 2 Technology Edition, Version 1.4.2
- IBM SDK for z/OS, Java 2 Technology Edition, Version 5

The support for Java 5 was added to CICS TS V3.2 through PTF PK59577. However, only one level can run in a CICS region at a time.

With Java 5 came a simplified shared class cache mechanism, enhanced garbage collection (GC), and optimized JIT compiler technology.

Also introduced in CICS TS V3.2 were many features for managing JVMs inside CICS, which included usability enhancements to the JVM profile and properties files and the ability to initialize and terminate JVMs by profile.

1.2.6 CICS Transaction Server V4.1

CICS TS V4.1 supports the JVM provided by the IBM 31-bit SDK for z/OS, Java Technology Edition, Version 6.0.0 and Version 6.0.1. To use Version 6.0.1, you must apply PTF PM38397. CICS TS V4.1 does not support the 64-bit version of the Java SDK.

The JVM server model and the JVMSERVER resource were first introduced in CICS TS V4.1. The JVM server model provided a key concept in the evolution of Java in CICS. That concept was of a single, long running JVM with the ability to run multiple concurrent CICS tasks as threads of execution within the JVM. In fact, the JVM server can handle up to 256 threads simultaneously.

Each CICS task that is required to run Java is switched onto a new type of (OTE) task control block (TCB), called a T8. This T8 TCB is a hybrid TCB providing both CICS capabilities (EXEC CICS) and Java capabilities (as a Java thread).

The JVMSERVER resource manages its own set of T8 TCBs taken from a region-wide pool of T8 TCBs. A “thread-limit” field on the resource provides a way to control concurrent thread numbers, while the underlying JVM is configured with a JVM profile. The Language Environment enclave in which the JVM runs can also be configured by a Language Environment runtime options file, which is again referenced by the JVMSERVER resource.

However, at this version of CICS, the JVM server relies on existing Java development practices Java ARchives (JARs) on a class path. It does not provide any tooling for application development or deployment, and there is no application lifecycle control (without restarting the JVM server). Furthermore, to run a Java program in the JVM server requires the developer to link to a special
program (DFHSSJJJ) and provide configuration via a channel and container interface.

1.3 CICS Transaction Server V4.2

CICS TS V4.2 (released in 2011) brought significant enhancements to the JVM server. An entire Java development platform was added with the CICS Explorer SDK. Application lifecycle management, deployment, and packaging were provided through adoption of the OSGi specification and the use of CICS bundles. Scalability was enhanced and storage limitations relieved yet further, with a move to 64-bit JVMs. Notably, Java application linkage was achieved with the enhancement of CICS program definition support for JVM server. We describe many of these aspects in the following chapters.

1.3.1 64-bit JVM support

CICS TS V4.2 uses the IBM 64-bit SDK for z/OS, Java Technology Edition, Version 6.0.1. It no longer supports a 31-bit version of the SDK.

JVMs of this type are running in addressing mode (AMODE)(64) and can address up to 16 exabytes of data. This provides the advantage that JVM stack and heap storage can be placed above the bar (that is, beyond the 31-bit or 2 GB address space). Then, many more JVMs can operate in the same (larger) address space.

Previously, 31-bit JVMs could address only their stack and heap storage below the bar, which has a much smaller address range. This configuration restricts the number of JVMs that can be run in a single CICS region’s address space.

**Attention:** The move to 64-bit JVMs greatly increases the scalability of a single CICS region. We now have the potential for hundreds of JVMs in a single region; whether that be JVM servers, or pooled JVMs.

1.3.2 Java virtual machine server and pooled JVMs

The introduction of the more powerful, and more standard JVM server model in CICS TS V4.1 meant that the pooled JVM model was no longer strategic to CICS. Pooled JVMs are still available and supported in CICS TS V4.2. However, they are functionally stable and have been removed entirely in CICS TS V5.1. The JVM server is the strategic direction for running Java applications in CICS.
Of key importance to the future of CICS is the more standard Java behavior of the JVM server model. By supporting a multi-threaded architecture, and running Java applications in the same way as they would run on a distributed platform, we open the doors to a multitude of potential exploiters.

The benefits however, are not exclusively for new workload. Migration of applications from pooled JVM to JVM server is an important topic that is covered extensively later in the book. Most of that exercise is a “repackage and redeploy” operation. However, it should be understood that unlike the pooled JVM environment, your application does not get a “new” JVM for each task. Your Java code should therefore be threadsafe, and particular care should be made around the use of static Java variables.

Figure 1-2 shows a comparison of a JVM server and pooled JVMs in CICS. JVM server setup and configuration are described extensively in Chapter 3, “Setting up a Java virtual machine server” on page 41.

### JVM server vs. existing pooled JVM Java support

<table>
<thead>
<tr>
<th>Single JVM – serves many tasks (reduced storage)</th>
<th>Pool of JVMs – each serves only a single task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrent, multi-threaded, up to 256 threads per JVM server</td>
<td>Java program isolation</td>
</tr>
<tr>
<td>T8 (CICS key)</td>
<td>J8 (CICS key), J9 (User key)</td>
</tr>
<tr>
<td>MAXTHRDTCBS (automatically calculated), up to max of 1024 per region</td>
<td>MAXJVMTCBS, SIT parm</td>
</tr>
<tr>
<td>More standard server model (+ data sharing)</td>
<td>Difficult, convoluted to share data and state</td>
</tr>
<tr>
<td>Dynamic update and replacement of modules</td>
<td>JVMs must be restarted to affect changes</td>
</tr>
</tbody>
</table>

Figure 1-2  JVM server versus pooled JVM

#### 1.3.3 Open Services Gateway initiative

Before CICS TS V4.2, class management was heavily focused on the JVM class path and the process of loading classes was often relatively expensive because
of the large number of locations that needed to be searched. If the JVM was required to load a class, it scanned the directories and JAR files indicated on the class path for that class. If a change to the class path was made, or even just a change to existing application code, the JVM had to be restarted to enable reload of the new classes.

For pooled JVMs in CICS, a PHASEOUT option of the PERFORM JVMPOOL command was available to ease the restarting of JVMs in the pool. However, when you are dealing with single, long running JVMs, actioning a JVM restart results in application downtime.

Therefore, a new model was required where applications could be developed into components and scoped, where dependencies could be made explicit and resolved ahead of time, where applications could be dynamically refreshed, or versioned and run side-by-side in the same JVM. This model is *Open Services Gateway initiative (OSGi)*, and it is provided by the OSGi Alliance (http://www.osgi.org).

The OSGi framework is a module system and service platform for the Java programming language that implements a complete and dynamic component model. Applications or components (in the form of bundles for deployment) can be remotely installed, started, stopped, updated, and uninstalled without requiring a JVM restart. The service registry allows bundles to detect the addition of new services, or the removal of services, and adapt accordingly.

In CICS, OSGi vastly improves the deployment and management lifecycle of Java applications. As of CICS TS V4.2, JVM servers support the deployment of OSGi bundled applications into the JVM server. A full description of the OSGi standard and how it is implemented by CICS TS V4.2 can be found in Chapter 2, “Open Services Gateway initiative overview” on page 19.

### 1.3.4 CICS Explorer software development kit

The CICS Explorer is an Eclipse-based system management tool that was released initially as a SupportPac back in 2008. It is an Eclipse-based graphical user interface (GUI) application that runs on the Windows or Linux operating systems and connects to either a CICSPlex SM WUI server or to a single CICS region. It is a no-charge downloadable application that can be downloaded from http://www.ibm.com/software/htp/cics/explorer/download/. The Explorer can be extended with plug-ins for CICS Performance Analyzer, CICS Interdependency Analyzer, CICS Configuration Manager, and CICS Transaction Gateway. A plug-in for IBM OMEGAMON® XE for CICS is also available as a Tivoli OPAL contribution.
The CICS Explorer SDK is an Eclipse-based framework for developing extensions to the CICS Explorer. Importantly, from a Java programming perspective, it also provides support for developing Java applications to run in any supported release of CICS. It provides support for JCICS and the creation and deployment of CICS Java OSGi applications. OSGi is covered in more detail in Chapter 2, “Open Services Gateway initiative overview” on page 19.

1.3.5 Support for Axis2 technology

CICS Transaction Server has provided support for CICS now for several releases. CICS is able to act as either a web services provider or as a web services requester. To do this, CICS implemented the pipeline concept whereby service requests, inbound or outbound, go through a series of handlers to process the web service being called or to call another web service. The pipeline is the bridge between the web service request, which is typically in XML, and the CICS resources needed to process or make the request.

These pipelines in CICS call a series of handler programs to process the incoming or outgoing SOAP messages. In previous versions of CICS, none of these handlers were written in Java. CICS TS V4.2 supports use of the Axis2 Java based SOAP engine to process web service requests in provider and requester pipelines. As Axis2 uses Java, the SOAP processing is eligible for offloading to the System z Application Assist Processor (zAAP) it runs in a JVM server.

Axis2 is an open source web services engine from the Apache foundation and is provided with CICS to process SOAP messages in a Java environment. You can opt into using Axis2 by adding a Java SOAP handler to your pipeline configuration file and creating a JVM server to handle the Axis2 processing. When CICS is a service provider, the Java based terminal handler uses Axis2 to parse the SOAP envelope for a request message. You can use header processing programs to process any SOAP headers associated with the SOAP message. Axis2 also constructs the SOAP response message.

1.3.6 Enterprise JavaBeans and pooled Java virtual machine removal

CICS TS V4.2 is the last release of CICS TS to support the JVM pool infrastructure. Therefore, we advise all users to deploy their Java applications to the CICS JVM server run time. The CICS TS V4.2 announcement letter issues the following statement of direction:

Statement of direction: IBM intends a future release of CICS TS to discontinue support for both: (i) session beans using Enterprise JavaBeans (EJB); and (ii) the Java pool infrastructure. Customers are encouraged to migrate Java applications
to the new JVM server infrastructure. For more information, see the following web page:


Further to that announcement, EJB and pooled JVM support have been removed in CICS TS V5.1.

1.4 Java class library for CICS as an API

The CICS API is an important foundation for developing applications in CICS. It is the mechanism by which programs running in CICS should access resources such as files, databases, or other programs. Traditionally, the CICS API has been available to high-level programming languages such as COBOL, PL/I, and also assembler. Similarly, in Java, the Java programmer is able to call the Java API equivalent to the familiar EXEC CICS commands used in high-level programming languages. For Java, this API is known as the JCICS interface.

Access to the JCICS interface is provided through an OSGi bundle called com.ibm.cics.server.jar. This bundle is essentially a JAR file with more metadata, and as such, it can be used in both the pooled JVM environment and JVM server. The bundle provides the necessary classes to access and manipulate CICS resources and to integrate Java programs with other programs written in any other language supported by CICS. The JCICS class library can be used in integrated development environment (IDE) development tools such as Eclipse or IBM Rational® Developer for System z to develop programs before deploying them to CICS for testing and execution.

The JCICS API is described extensively in Chapter 7, “The Java CICS application programming interface” on page 187.

1.4.1 Interfacing with language structures

One of the intentions of this book is to describe how the Java language is used in CICS Transaction Server. CICS is a well established mature environment for which there exists millions of lines of application code that is written by using high-level languages such as COBOL or PL/I. It is inevitable therefore, that there will be a need to interact with these high-level languages. This brings the challenge that Java programs must be able use structured data from those high-level languages. For example, Java programs might link to a COBOL program passing a commarea (or a channel) where the data inside the channel or commarea is defined in a COBOL copybook. These copybooks are typically
used where Virtual Storage Access Method (VSAM) file records and database records are accessed. The copybooks must be converted into Java constructs that Java programs can easily manipulate.

CICS supports two tools, which can be used to import copybooks so that you can use the structured data from other programming languages in your Java programs.

CICS supports Java artifacts from the following importers:

- Data binding beans from the Java EE Connector Architecture (J2C) tools in Rational Application Developer and Rational Developer for System z
- Records from the IBM JZOS Batch Toolkit for z/OS SDK

A more detailed explanation and demonstration of these tools can be found in Chapter 8, “Interacting with structured data” on page 253.

### 1.5 64-bit storage exploitation

Since CICS TS V3.2, there has been a gradual exploitation of the 64-bit capabilities of the IBM zSeries® architecture. Before CICS TS V3.2, CICS address spaces were restricted to 31-bit. This relatively small address space (by modern computing standards), imposed storage limitations. Scalability and availability were constrained, yet many customers learned to work around these limits by scaling outward with more CICS regions. The key changes associated with 64-bit exploitation in CICS are as follows:

- CICS TS V3.2 began the process of using 64-bit storage in CICS by using above-the-bar storage for channels and containers.
- CICS TS V4.1 followed on by extending support for 64-bit to the events architecture, XML parsing, and to CICSPlex SM. However, CICS TS V4.1 was still largely a 31-bit environment.
- CICS TS V4.2 has made significantly more changes to use 64-bit addressing than any of its predecessors. Major changes were made to the CICS domain architecture to use the underlying IBM z/Architecture® for 64-bit addressing. There are many more areas of CICS now using 64-bit addressability, including the CICS JVM.

#### 1.5.1 Java 64-bit support

One of the areas using 64-bit addressability is Java. As stated earlier, CICS TS V4.2 supports the *IBM 64-bit SDK for z/OS, Java Technology Edition, Version 6.0.1*. This support is for both pooled JVMs and JVM servers.
CICS will issue an abnormal end of task (abend), ASJJ, if JAVA_HOME points to any version of Java except 64-bit Java 6.0.1. CICS message DFHSJ0900 will also be written to the log. See Example 1-1.

Example 1-1  Illegal Java version message

DFHSJ0900 03/10/2011 11:00:07 IYK2ZIK1 Illegal Java version. CICS requires Java version 6.0.1 but has found Java version 1.5.0

Similarly, if an attempt is made to reference a 31-bit JVM, abend ASJD will be issued and message DFHSJ503 will be written to the log. See Example 1-2.

Example 1-2  Attempt to reference 31-bit Java message

DFHSJ0503 03/10/2011 10:50:21 IYK2ZIK1 DFHJVMPR Attempt to load DLL libjvm.so has failed. Runtime error message is EDC5253S An AMODE64 application is attempting to load an AMODE31 DLL load module. (errno2=0xC40B0013)

1.5.2 31-bit JVM limitations

Before CICS TS V4.2, the storage for a CICS address space was limited to 2 GB in size. Regarding Figure 1-3 on page 16, if you look at storage above the 16 MB line, the storage available is allocated in a standard way. Global storage is allocated at the bottom of the extended storage, above the 16 MB line and above the extended user region. This is the area of the address space that is used by CICS for its extended dynamic storage areas (EDSAs). The size of this area will be determined by the EDSALIM parameter in the CICS system initialization table (SIT). Next from the top of the address space, the operating system takes an area for extended local system queue area (LSQA), scheduler work area (SWA), and subpools 229/230.
As shown in Figure 1-3, the remaining free area lies in-between the storage allocated at the bottom of the extended area and that allocated from the top of the region. It is from this area that the Language Environment enclave storage that is required by the JVM is allocated.

![Diagram of 31-bit address space]

Immediately, you can see that the size of the free area is constrained by the Extended LSQA/SWA/229/230 area above and the Extended user region below. This in turn will limit the number of JVMs that a 31-bit address space can run simultaneously. Typically, this is only a handful of JVMs per CICS region. If CICS attempts to start another JVM and there is not sufficient storage available in the free extended storage area, you see this message:

DFHSJ0516 11/04/2012 11:24:14 EPRED1 JVMPROF An attempt to create a Java Virtual Machine using the JNI has failed. See the JVM's STDERR log for further details.

1.5.3 64-bit JVM advantages

CICS TS V4.2 supports the IBM 64-bit SDK for z/OS, Java Technology Edition, Version 6.0.1. It is able to use the vast area addressable above the 2 GB bar, removing the limitations that we described previously with the free extended area below the 2 GB bar. The address space can potentially extend to 16 exabytes.
(which is 1,000,000,000 gigabytes or 1,000,000 terabytes). See Table 1-1 for a definition of an exabyte and other smaller size definitions.

Table 1-1 byte definitions

<table>
<thead>
<tr>
<th>Name (with abbreviation)</th>
<th>Decimal value</th>
<th>Binary value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kilobyte (KB)</td>
<td>$10^3$</td>
<td>$2^{10}$</td>
</tr>
<tr>
<td>megabyte (MB)</td>
<td>$10^6$</td>
<td>$2^{20}$</td>
</tr>
<tr>
<td>gigabyte (GB)</td>
<td>$10^9$</td>
<td>$2^{30}$</td>
</tr>
<tr>
<td>terabyte (TB)</td>
<td>$10^{12}$</td>
<td>$2^{40}$</td>
</tr>
<tr>
<td>petabyte (PB)</td>
<td>$10^{15}$</td>
<td>$2^{50}$</td>
</tr>
<tr>
<td>exabyte (EB)</td>
<td>$10^{18}$</td>
<td>$2^{60}$</td>
</tr>
</tbody>
</table>
The most important benefit that the 64-bit JVM provides is the ability to use virtual storage above the 2 GB bar to store Language Environment enclave data and the JVM heap and stack storage. This means that we can have potentially hundreds of JVMs in a single CICS region. This configuration can be seen in Figure 1-4.

![Figure 1-4 64-bit address space](image)

In addition to the possibility of more Language Environment enclaves, we can also have much larger enclaves, which support larger JVM heaps and can help reduce the impact of garbage collection within the JVM.
Open Services Gateway initiative overview

This chapter introduces the Open Services Gateway initiative (OSGi) and the OSGi architecture. We provide an overview of OSGi support in CICS, and demonstrate the use of CICS bundles as a way to deploy and manage Java applications. The chapter concludes with a look at the new CICS system programming interface (SPI) commands used to interact with OSGi bundles and OSGi services.

OSGi is essentially a mechanism allowing component-based development of Java applications. The component-based applications can be dynamically added, removed, and assigned a version number within the OSGi run time. More benefits are achieved when use is made of the OSGi service registry. Applications can dynamically publish the services that they implement, or discover service instances to use (find-bind-publish).
2.1 Open Services Gateway initiative architecture

At the heart of the OSGi specification is the OSGi architecture. This architecture is underpinned by a set of conceptual layers (as seen in Figure 2-1).

![Layers of the OSGi architecture](image)

Within this architecture the term *bundles* refers to an OSGi component, where one or more bundles form an OSGi application. OSGi bundles are installed into an OSGi framework, and it is the OSGi framework that provides the execution environment on which the OSGi bundles run. The execution environment layers define which classes and methods are available in a specific platform, and the Security layer implements OSGi security at any layer level.

**OSGi bundles**

OSGi bundles are essentially Java ARchive (JAR) files with extra headers placed in the MANIFEST.MF file. The headers describe the bundle (such as *name* and *version*), and explicitly describe the interaction of this bundle with other bundles. By default, Java packages within OSGi bundles are not exposed outside of that bundle. Therefore, explicit declarations (Import-Package and Export-Package headers) are required to allow a bundle to interface with other bundles.

**Services layer**

The Services layer of the OSGi architecture manages the service registry. An OSGi bundle might choose to register its services with this registry, or even use the registry to find an implementation of a service. Services *decouple* a capability from any particular implementation and provide the dynamism of OSGi. Services can be registered programmatically, or even through metadata (such as Declarative Services, or the Blueprint Container specification).
**Life Cycle layer**

The Life Cycle layer manages the lifecycle of bundles in an OSGi framework. Bundles can be installed, started, stopped, and uninstalled. Unlike classic Java applications, bundle lifecycle management is dynamic. The Java virtual machine (JVM) does not need to be restarted at any phase of the application lifecycle. Classic Java application deployment requires the class path to be updated and the JVM to be restarted.

**Modules layer**

The Modules layer manages *modularity, bundle dependencies, and versioning*. These characteristics are provided through a new architecture of class loaders. Each bundle has its own class loader. Class sharing is realized at the package level through imports and exports declared in the manifest file. Unlike the standard Java class loader architecture, when a class must be loaded, the request is handled directly by the class loader for that bundle. The standard Java class loaders architecture is hierarchical and a class search algorithm delegates the load request to the parent loader first. This causes application classes to be searched in all of the three hierarchical loader levels, which is not as efficient as using the bundle’s single class loader, see Figure 2-2.

![Figure 2-2 OSGi bundle class loading](image)

Importantly, in OSGi, multiple versions of the same package can coexist in the same runtime environment. This is made possible by a combination of features:

- Isolation of bundle class loaders
- The ability to specify package versions in the *Export-Package* header
- The ability to use version ranges in the *Import-Package* header
Bundle *constraints*, declared by the Import-Package manifest header, are normally resolved when the bundle is installed. This minimizes the chances of a ClassDefNotFound exception at run time.

### 2.2 OSGi implementation in CICS

CICS Transaction Server (TS) V4.2 provides four important aspects to its OSGi support: the OSGi run time, the CICS OSGi plug-ins, the Explorer software development kit (SDK), and CICS bundles.

**OSGi run time**
The OSGi runtime environment is provided by the OSGi framework running in a JVM server, and is started when the JVM server is enabled.

**Explorer SDK**
To complement the run time, CICS provides the Explorer SDK to create and deploy OSGi applications. If you are not familiar with the CICS Explorer SDK, a CICS bundle project serves as the main artifact of development. It references the application projects that you want to deploy to CICS. In our examples, those application projects are the OSGi bundles (or plug-in projects as they are better known in Eclipse-based tools). The terms *OSGi bundle* and plug-in project are used interchangeably.

**CICS OSGi plug-ins**
CICS provides three OSGi plug-ins, which contain the Java packages to be used by CICS application programs (Table 2-1). These are provided in the Explorer SDK, and within the CICS run time, and must be set as dependencies within the OSGi bundle manifest when creating your own OSGi plug-in projects.

<table>
<thead>
<tr>
<th>Table 2-1</th>
<th>CICS OSGi plug-ins</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OSGi bundle</strong></td>
<td><strong>JAR file</strong></td>
</tr>
<tr>
<td>com.ibm.cics.server</td>
<td>com.ibm.cics.server.jar</td>
</tr>
<tr>
<td>com.ibm.cics.server.pipeline</td>
<td>com.ibm.cics.server.pipeline</td>
</tr>
<tr>
<td>com.ibm.record</td>
<td>com.ibm.record.jar</td>
</tr>
</tbody>
</table>

The principal plug-in is `com.ibm.cics.server`, which provides the Java class library for CICS (JCICS) interfaces. Because each version of CICS has different
JCICS interfaces, the different versions of JCICS have been packaged within the CICS Explorer as different OSGi package versions, using the schema `major.minor.micro` (Table 2-2). You can see that in each release of CICS TS since V3.1, the minor version number has been incremented, and more service fixes can potentially increment the micro versions.

Table 2-2   OSGi package versions

<table>
<thead>
<tr>
<th>CICS TS version</th>
<th>com.ibm.cics.server OSGi package version</th>
</tr>
</thead>
<tbody>
<tr>
<td>CICS TS V3.1</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>CICS TS V3.2</td>
<td>v1.100.0</td>
</tr>
<tr>
<td>CICS TS V4.1</td>
<td>v1.200.0</td>
</tr>
<tr>
<td>CICS TS V4.2</td>
<td>v1.300.0</td>
</tr>
</tbody>
</table>

For details about how to use the CICS OSGi packaging versions in your Java application, see 6.1.3, “Edit the plug-in manifest file” on page 168.

**CICS bundles**

Finally, there are CICS bundles, introduced in a previous release of CICS as a way to package Atom Services and Events. The CICS BUNDLE resource has been extended in CICS TS V4.2 to provide a convenient mechanism for packaging, deploying, and offering lifecycle control of OSGi bundles.

Furthermore, multiple OSGi bundles might be in a single CICS bundle. It is important to recognize the use of the term *bundle* as applied to OSGi and to CICS. A CICS bundle is a CICS resource that is used for managed CICS artifacts that are on a zSeries file system (zFS) (including OSGi bundles). An OSGi bundle is specifically a Java archive conforming to the OSGi specification.

**Note:** Care should also be taken when describing CICS bundle projects regarding the Explorer SDK, and CICS BUNDLE resources within CICS. The CICS bundle project is not a valid mapping of a CICS bundle until it is exported to zFS by the Explorer SDK.

When you have an OSGi bundle ready to deploy to CICS, the CICS Explorer SDK provides two useful wizards. One wizard associates the Java plug-in projects with the CICS bundle project. The second wizard deploys the CICS bundle project (including the referenced OSGi bundles) to zFS.
The CICS bundle project is exported to a bundle directory on zFS where a directory structure is created that reflects the structure expected by the BUNDLE resource.

**The CICS bundle directory structure**

The bundle directory contains both metadata, and the JAR files (OSGi bundles) encapsulating the compiled runtime code. The file `cics.xml` is the CICS bundle manifest. It describes the contents, or bundleparts, of the CICS bundle. Each bundlepart file is itself a descriptor, and in our case describes the actual OSGi bundles. See Figure 2-3.

![Figure 2-3 Structure of a CICS bundle directory containing an OSGi bundle](image)

After a CICS bundle project has been deployed to zFS, a corresponding BUNDLE resource might be defined in CICS. The BUNDLEDIR attribute of this resource should point to the same location on zFS where the CICS bundle project was deployed.

Deployment of an actual OSGi bundle into a JVM server is driven by the installation of a CICS BUNDLE resource in CICS. The lifecycle of the OSGi bundle within the CICS bundle is controlled by the lifecycle of the CICS bundle and its bundle parts.

**Migrating existing Java projects**

If you are concerned about migration of existing Java projects, Eclipse allows existing Java projects to be converted into plug-in projects. It is a fairly simple task to convert existing CICS Java programs into OSGi bundles by using the right-click option **Configure → Convert to Plug-in Project**.
2.3 Artifacts

You will encounter several new file artifacts when working with OSGi bundles and CICS bundles. It is useful to understand their relevance to CICS and their operation regarding the applications you will develop.

2.3.1 Metadata files

We mentioned previously that an OSGi bundle is just a JAR file with some additional OSGi defined headers in the MANIFEST file. These headers are the metadata that when interpreted by the OSGi runtime, turn the humble JAR file into a much more conceptually powerful entity: the OSGi bundle. These headers determine the scope of the application, its version, its dependencies, and any function exposed to other components in the system.

Similarly, a CICS bundle has a manifest file (cics.xml). The CICS manifest file contains metadata describing the CICS bundle and its component bundle parts.
Figure 2-4 shows the contents of a directory into which a CICS bundle project that contains a single OSGi bundle has been deployed. The CICS manifest file (cics.xml) is located under the META-INF subdirectory. The structure of the XML in the CICS bundle manifest file is defined by the schema bundle.xsd, which is in the /usr/lpp/cicsts/cicsts42/schemas/bundle.

```xml
<?xml version=“1.0” encoding=“UTF-8” standalone=“yes”?>
    <meta directives>
    </meta directives>
</manifest>
```

Figure 2-4  CICS bundle manifest file defining an OSGIBUNDLE resource

Each `define` element in the CICS bundle manifest file defines a resource contained in the CICS bundle. The `type` and `name` attributes of the `define` element provide the name and type of the resource. The `path` attribute contains the metadata file name for that resource.

The CICS manifest file in Figure 2-4 defines a resource called `jcics` of type OSGIBUNDLE, and the metadata file `jcics.osgibundle` describes that particular component of the CICS bundle. In CICS, the file extension of the metadata file indicates the type of resource defined. For CICS osgibundle parts, the metadata file extension is .osgibundle, although the content is defined by the schema in file OsgiBundle.xsd, located under the /usr/lpp/cicsts/cicsts42/schemas/OSGi directory.

If you look inside the osgibundle metadata file, you see three attributes:

- **symbolicname**: The OSGi bundle symbolic name, as defined by the OSGi specification. CICS Explorer SDK retrieves this field from the Bundle-SymbolicName header of the OSGi bundle manifest file.
Chapter 2. Open Services Gateway initiative overview

- **version**: The OSGi bundle version, as defined by the OSGi specification. CICS Explorer SDK retrieves this field from the Bundle-Version header of the OSGi bundle manifest file. The format used is `major.minor.micro`, where major, minor, and micro are numeric fields.

**Restriction**: The OSGi specification allows a further version qualifier in the format `major.minor.micro.qualifier`. However, CICS does not support the qualifier field. If this field is used in an OSGi bundle that is imported into the CICS bundle, the CICS Explorer SDK will mark the osgibundle file with an error and the qualifier must be removed in order for the CICS bundle project to compile.

- **jvmserver**: The name of the JVMSERVER resource into which the OSGi bundle will be installed.

![Metadata file describing an OSGi bundle resource in a CICS bundle](image)

Figure 2-5 shows the content of an osgibundle file that defines an OSGi bundle resource whose symbolic name is `com.ibm.cics.server.examples.jcics`. The version is `1.0.0`, and the JVMSERVER resource name is `DFH$JVMS`.

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<osgibundle symbolicname="com.ibm.cics.server.examples.jcics" version="1.0.0" jvmserver="DFH$JVMS"/>

Figure 2-5 Metadata file describing an OSGi bundle resource in a CICS bundle

Figure 2-5 shows the content of an osgibundle file that defines an OSGi bundle resource whose symbolic name is `com.ibm.cics.server.examples.jcics`. The version is `1.0.0`, and the JVMSERVER resource name is `DFH$JVMS`.
**OSGi bundle manifest headers**

Take a closer look at the OSGi bundle `MANIFEST.MF` file. When a plug-in project is created, this manifest file is created under the `META-INF` directory of the plug-in project. The OSGi specification defines a set of headers, which describe the characteristics of an OSGi bundle (Figure 2-6).

![Figure 2-6 OSGi bundle manifest file](image)

The following key headers are used in the manifest:

- **Bundle-ManifestVersion**
  
The bundle manifest version. This is set to 2 by the CICS Explorer SDK, which is the manifest version for bundles implementing the OSGi R4 spec.

- **Bundle-Name**
  
  A textual identifier for the bundle. This value is set from the Name field in the new plug-in project wizard.

- **Bundle-SymbolicName**
  
  Together with the Bundle-Version, this uniquely identifies a bundle in an OSGi framework. This field is set from the ID field in the new plug-in project wizard.
- **Bundle-Version**
  The bundle version in the form of `major.minor.micro.qualifier`. This field is set from the version field in the new plug-in project wizard. CICS does not support the qualifier and you must remove it in the new plug-in project wizard.

- **Bundle-RequiredExecutionEnvironment**
  The runtime environment denotes the packages that are required for the bundle.

- **Import-Package**
  Lists the packages that the bundle depends on. This header is preferred over the use of the Require-Bundle header.

- **Require-Bundle**
  This specifies that all exported packages from another bundle must be imported.

**CICS defined manifest headers**
For OSGi bundles deployed to CICS, CICS allows an extended header, called the `CICS-MainClass` header. The CICS-MainClass header is used to specify a list of classes that act as entry-points and can be specified on a CICS program definition. Each of these entry-point classes must contain a main method, and be within the OSGi bundle. CICS uses the OSGi service registry to make these entry-points available outside the normal scope of the OSGi framework. A service lookup is performed when the CICS program is started. It is this mechanism that gives the initial CICS linkage to Java programs.

Figure 2-7 shows the list of registered OSGi services.
The CICS-MainClass header also allows aliases to be specified. These aliases map the package-qualified class name to something of the user’s choice. The alias would generally be used to associate the package/class with something more familiar to the user (an application name perhaps), or as a way to provide multiple versions of the same application. See Chapter 4, “Deploying from CICS Explorer SDK” on page 71 for more details.

Figure 2-8 shows how to specify an alias. If no aliases are specified, the package-qualified class name will be used.

![Figure 2-8 Use of aliases in CICS-MainClass header](image-url)
Figure 2-9 shows services published by two different versions of the same CICS bundle (or application), one of which uses aliases for the service names.

### 2.4 OSGi application management

In CICS, there are conceptually two different types of OSGi bundles: Application OSGi bundles, and middleware OSGi bundles. The application OSGi bundles are managed as part of the CICS bundle lifecycle. Middleware OSGi bundles are installed during JVM server initialization and follow the lifecycle of the JVM server. Middleware OSGi bundles are typically updated less often. They are made available for the lifetime of the OSGi framework, and offer function that other applications build upon; for example, a database driver.

#### 2.4.1 OSGi bundle installation

OSGi bundles are installed when their “containing” CICS BUNDLE resource is installed into CICS. Part of this installation process is to find the OSGi bundle on the zFS file system (specified on the BundleDir attribute) and to inject it into the OSGi framework. To speed up the file search, the JVM server looks for file names starting with `symbolicname_version`, which is the file name pattern used by the CICS Explorer SDK during the OSGi bundle export process. If no match is found, the search is extended to look at any JAR file in the bundle directory. A
match is determined when the SymbolicName and Version headers in the bundle's manifest file match that specified by the osgibundle metadata file (part of the CICS bundle descriptor). If you are not using Explorer SDK to produce your OSGi bundles, you might want to follow this naming convention to provide optimal matching.

The installation of CICS bundles is a two-phase process and this maps well to the OSGi installation process. As a result of the CICS bundle installation, any contained OSGi bundles are installed into the OSGi framework allowing the framework to read their metadata and determine if dependencies can be resolved. The second phase of installation is to enable the CICS bundle, or start the OSGi bundles. If the OSGi bundles start successfully, you can be confident that their dependencies have been resolved and you will not receive ClassDefNotFound Exceptions at run time.

When an OSGi bundle has been started, CICS is notified and will determine whether any CICS-MainClass headers are present in the manifest file. For each CICS-MainClass defined, a corresponding OSGi service is registered for that package-qualified class combination (optionally with the alias name supplied).

If there are OSGi bundles with the same CICS-MainClass already installed, the service is installed as a duplicate and considered inactive by CICS, meaning it cannot be linked to. If the original OSGi bundle is then stopped, causing its CICS-MainClass service to be deactivated, the duplicate service is “promoted” to active and becomes the new target of any CICS program links.

**Tip:** Careful use of aliases provides the ability to deploy parallel versions of the same libraries and applications into the same JVM. Use of duplicate services also allows a smooth application upgrade path, when new code is deployed, providing a simple mechanism to provide continuity of the service.

**Installation considerations**

When a CICS bundle containing OSGi bundle parts is installed and the target JVM server is not installed, the installation fails and the bundle part is put into an unusable status. The installation cannot be recovered because CICS cannot leave the installation process hanging while the JVM server might or might not become installed. You **must** discard the bundle before further installation attempts. The intuitive action to re-enable the bundle after the JVM server has been installed, will not have the wanted effect.

If the JVM server is installed, but is in a disabled status, the installation will return success, and the OSGi bundle is put on an internal chain ready to be installed as the JVM server becomes enabled.
Dependencies between OSGi bundles defined within the same CICS bundle are handled by CICS, and the ordering of OSGi bundle definitions in a CICS bundle manifest file is not relevant. When a CICS bundle is installed as enabled, the OSGIBUNDLE bundle parts are installed. They are only enabled when all the bundle parts have been installed (ensuring dependencies can be resolved).

Similarly, if the OSGi bundle installation is postponed because the JVM server is disabled, at the point the JVM server becomes enabled all the OSGi bundles will be installed as phase 1. Then, all the OSGi bundles are enabled as phase 2.

For more information about the use of aliases for versioning, see 4.6.4, “Updating Java applications” on page 127.

### 2.4.2 OSGi bundle status transition

There are various stages within the lifecycle of an OSGi bundle. For reference, they are described here. Figure 2-10 shows how an OSGi bundle changes its status during its lifecycle in the OSGi framework.

![OSGi bundle status transition diagram](image)

The **INSTALLED** status means that the bundle has just been installed and no other status transitions have happened yet.

The **RESOLVED** status means that the OSGi bundle has been installed and its constraints have been successfully resolved, or that the bundle was previously active and has been stopped.

The **ACTIVE** status means that the bundle has been started successfully.
The remaining states are transient states, except for the \textit{STARTING} state for a bundle with lazy activation. A lazily activated bundle will remain in the \textit{STARTING} state until any one of its classes is loaded.

In Table 2-3, after a CICS bundle is enabled, you see the status of the associated components, for example, the CICS resources, the OSGi bundle, and OSGi service.

\begin{table}[h]
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{State after enabling the CICS BUNDLE resource} & \textbf{CICS bundle} & \textbf{CICS bundle part} & \textbf{OSGi bundle} & \textbf{OSGi service} \\
\hline
Bundle enabling completes successfully & ENABLED & ENABLED & ACTIVE & ACTIVE \\
\hline
Bundle enabling completes, but duplicate OSGi service & ENABLED & ENABLED & ACTIVE & INACTIVE \\
\hline
Bundle enabling completes, but OSGi bundle has lazy activator & ENABLED & ENABLED & STARTING & N/A \\
\hline
OSGi bundle failed to start & DISABLED & DISABLED & INSTALLED & N/A \\
\hline
OSGi bundle failed to install & DISABLED & UNUSABLE & N/A & N/A \\
\hline
\end{tabular}
\caption{Status of components after enabling a CICS bundle}
\end{table}

\textbf{Note:} When the OSGi service state is shown as inactive, it is only inactive to CICS, not the OSGi service registry. The service will be installed and available. However, properties that describe it will indicate to CICS that it cannot be used.

At the higher level, a transition in JVM server status can affect the OSGi bundle lifecycle. For example, if the JVM server is disabled and then discarded, the BUNDLEPART resources become UNUSABLE and the corresponding CICS BUNDLE status becomes DISABLED. Table 2-4 lists some of the most common scenarios that can happen in the JVM server lifecycle and shows the status of the resources impacted by it.

\begin{table}[h]
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{JVM server action} & \textbf{Resource states} \\
\hline
\textbf{CICS bundle} & \textbf{CICS bundle part} & \textbf{OSGi bundle} & \textbf{OSGi service} \\
\hline
JVM server re-enabled and previous OSGi bundle status was active & ENABLED & ENABLED & ACTIVE & ACTIVE or INACTIVE \\
\hline
\end{tabular}
\caption{Status of components after JVM server lifecycle operations}
\end{table}
The current state of all the resources in this section can be viewed in CICS Explorer. The *OSGi bundles* and *OSGi Services* views in particular provide lots of useful information that is taken directly from underlying OSGi framework. Figure 2-11 shows the CICS Explorer views related to OSGi resources.

<table>
<thead>
<tr>
<th>JVM server action</th>
<th>Resource states</th>
</tr>
</thead>
<tbody>
<tr>
<td>JVM server is disabled while the OSGi bundle is active</td>
<td>DISABLED</td>
</tr>
<tr>
<td>JVM server is discarded</td>
<td>DISABLED</td>
</tr>
<tr>
<td>JVM server is installed after the BUNDLE installation</td>
<td>DISABLED</td>
</tr>
</tbody>
</table>
2.4.3 OSGi bundle interdependency management

Consider the scenario where you have two CICS bundles, Bundle-A and Bundle-B, and each CICS bundle contains an OSGi bundle. If the OSGi bundle in Bundle-A has a dependency on the OSGi bundle in Bundle-B, Bundle-B must be installed first, if Bundle-A is to enable correctly. This can be handled by ordering the System Programmer’s knowledge of their installation order. Or, if the OSGi bundle in Bundle-B is shared among other bundles, it can be added to the middleware bundle list by using the OSGI_BUNDLES variable in the JVM profile to ensure its availability at JVM server initialization.

However, there are scenarios where installing CICS bundles in a precise order does not provide a reliable activation method. For this reason, CICS extends the two-phase bundle activation process across all installed CICS bundles in the region, whenever bundles are activated through one of the following methods:

- CICS region warm start
- Bundles installed by grouplists at cold start
- Bundles enabled after a JVM server restart

This two-phase activation ensures that all OSGi bundles become installed before any of them are started, which ensures that dependencies between them are able to be resolved.

Note: CICS maintains a chain of CICS bundles associated with the JVM server resource even when it is disabled so that bundle activation will occur automatically when the JVM server is enabled. This means JVMSERVER resource must be fully discarded (or the CICS bundles discarded) to remove this association.

2.5 CICS commands for OSGi

Two new system programming interface (SPI) commands have been created in CICS TS V4.2 to return information about OSGi resources installed in a JVM server.

Restriction: These new OSGi SPI commands are not available via the CEMT interface. Therefore, we recommend that the CICS Explorer is used for OSGi bundle management operation.
2.5.1 EXEC CICS INQUIRE OSGIBUNDLE

This command can accept OSGi bundle symbolic name, version, and JVM server input parameters and return the properties of a matching OSGi bundle. Or, it can let the caller browse through the OSGi bundles installed on the specified JVM server.

These are the parameters passed or returned:

**OSGIBUNDLE** Specifies the 255-character symbolic name of the OSGi bundle.

**OSGIVERSION** Specifies the 255-character OSGi bundle version in the format used by OSGi: major.minor.micro.

**JVMSERVER** Specifies the eight-character JVM server name where the OSGi bundle is installed on.

**OSGIBUNDLEID** Returns the OSGi internal unique ID of the OSGi bundle.

**OSGISTATUS** Returns the CICS-value data area (CVDA) value of the OSGi bundle status. It can have one of the following values:

- **ACTIVE** The OSGi bundle started successfully; constraints were resolved and the activator method returned.
- **INSTALLED** The OSGi bundle has installed. Either no attempt to resolve the dependencies have been made or the OSGi framework failed to resolve the constraints.
- **UNINSTALLED** The OSGi bundle has uninstalled.
- **RESOLVED** The OSGi bundle has installed and its dependencies has been resolved.
- **STARTING** The OSGi bundle is being started. The activator start method has been called but not yet returned.
- **STOPPING** The OSGi bundle is being stopped. The activator stop method has been called but not yet returned.

**BUNDLE** Returns the eight-character name of the CICS bundle that installed the OSGi bundle.

**BUNDLEPART** Returns the eight-character name of the CICS bundle that installed the OSGi bundle.
CHANGEAGENT

Returns the CVDA value that identifies the agent that made the last change to the resource definition. The following list provides possible CVDA values:

CREATESPI

The resource was last changed by the EXEC CICS CREATE command.

CSDAPI

The resource definition was last changed by a CEDA, the transaction, or the programmable interface to DFHEDAP.

CSDBATCH

The resource definition was last changed by a DFHCSDUP job.

DREPAPI

The resource definition was last changed by a CICSplex SM BAS API command.

CHANGEAGREL

Returns a four-digit number of the CICS release that was running when the resource definition was last changed.

CHANGETIME

Returns an ABSTIME value that represents the time stamp when the resource definition was last changed.

CHANGEUSRID

Returns the eight-character user ID that ran the change agent.

DEFINESOURCE

Returns the eight-character source of the resource definition. The DEFINESOURCE value depends on the CHANGEAGENT value.

DEFINETIME

Returns an ABSTIME value that represents the time stamp when the resource definition was created.

INSTALLAGENT

Returns a CVDA value that identifies the agent that installed the resource. The possible values are as follows:

CREATESPI

The resource was installed by an EXEC CICS CREATE command.

CSDAPI

The resource was installed by a CEDA transaction or the programmable interface to DFHEDAP.

GRPLIST

The resource was installed by GRPLIST INSTALL.

INSTALLTIME

Returns an ABSTIME value that represents the time stamp when the resource was installed.

INSTALLUSRID

Returns the eight-character user ID that installed the resource.
2.5.2 EXEC CICS INQUIRE OSGISERVICE

Returns information about the OSGi service whose service ID has been provided as an input parameter on the specified JVM server. Alternatively, the user can browse through the OSGi services registered by the specified JVM server.

These are the parameters passed or returned:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSGISERVICE</td>
<td>Specifies the doubleword binary value representing the internal unique ID of the OSGi service.</td>
</tr>
<tr>
<td>JVMSERVER</td>
<td>Specifies the eight-character name of the JVM server.</td>
</tr>
<tr>
<td>SRVCNAME</td>
<td>Returns the 255-character name of the service.</td>
</tr>
<tr>
<td>SRVCSTATUS(CVDA)</td>
<td>Returns the CVDA value of the OSGi service status. It can assume one of the following values:</td>
</tr>
<tr>
<td>ACTIVE</td>
<td>The OSGi service is available.</td>
</tr>
<tr>
<td>INACTIVE</td>
<td>The OSGi service is not available because a duplicate service is active.</td>
</tr>
<tr>
<td>OSGIBUNDLE</td>
<td>Returns the 255-character symbolic name of the OSGi bundle that registered the service.</td>
</tr>
<tr>
<td>OSGIVERSION</td>
<td>Returns the 255-character version of the OSGi bundle that registered the service.</td>
</tr>
<tr>
<td>BUNDLE</td>
<td>Returns the eight-character name of the CICS bundle that installed the OSGi bundle.</td>
</tr>
<tr>
<td>BUNDLEPART</td>
<td>Returns the eight-character name of the CICS bundle part that installed the OSGi bundle.</td>
</tr>
</tbody>
</table>
Setting up a Java virtual machine server

The main focus of this chapter is to describe the configuration of a Java virtual machine (JVM) server. We also describe the lifecycle of the JVMSERVER resource and demonstrate the use of the various JVMSERVER system programming interface (SPI) commands.

This chapter divides JVM server configuration parameters into four groups:

- System initialization table (SIT) parameters
- Language Environment configuration parameters
- JVMSERVER resource definition parameters
- JVM profile parameters

We provide an explanation for each group of parameters and how they contribute to the configuration of the JVM server.
3.1 Java virtual machine server configuration

SIT parameters, Language Environment runtime options, parameters in the JVMSERVER resource definition, and JVM server profile options contribute to the configuration of a JVM server run time.

Figure 3-1 shows the different groups of configuration parameters in relation to the components that elaborate them.

3.1.1 System initialization table parameters

The system initialization table (SIT) parameters that must be considered when configuring a CICS region to run a JVM server are **USSHOME** and **JVMPROFILEDIR**.

**USSHOME**
The **USSHOME** SIT parameter specifies the zSeries file system (zFS) installation directory of the CICS UNIX System Services components. This is the location at which the CICS installation process has stored the UNIX System Services parts of CICS. The default value is `/usr/lpp/cicsts/cicst42`. You might need to consult with whomever installed CICS to determine the correct value.
If the directory pointed by USSHOME does not exist or USSHOME is set to **NONE**, the JVM server fails to start and error message DFHSJ0210 is written in the CICS log. A further error message, explaining the reason code, can be found either in the SYSPRINT or in a dynamically generated data definition (DD) name in the CICS log.

**JVMPROFILEDIR**

The **JVMPROFILEDIR** SIT parameter specifies the zFS directory where JVM server profiles are located. JVM server profiles are files that define the CICS and Java configuration of a JVM server. If not specified, CICS defaults the directory to `/usr/lpp/cicsts/cicsts42/JVMProfiles`.

CICS uses the JVMPROFILEDIR and the profile name specified in the JVMSERVER resource definition (see 3.1.2, “JVMSERVER attributes” on page 44) to locate the profile for a given JVM server. Figure 3-2 shows JVMPROFILEDIR and USSHOME SIT parameters used in the CICS regions used in the examples.

For more information about JVM profiles, see the following web page:

3.1.2 JVMSERVER attributes

Some of the attributes in the JVMSERVER resource definition contribute to the JVM server runtime configuration. In Figure 3-3 on page 45, the JVMSERVER Attributes panel of CICS Explorer shows the parameters of the JVMSERVER definition included in the Java sample group DFHOSGI.

The JVM profile attribute contains the file name of the JVM server profile located under the directory specified in the JVMPROFILEDIR SIT parameter. The JVM server profile defines sets of configuration parameters for the JVM. This is not an optional parameter. The sample JVMSERVER definition DFH$JVMS, shown in Figure 3-3 on page 45, uses the sample profile DFHOSGI provided by CICS (see 3.1.4, “JVM server profile for user applications” on page 47 for a description of the default profile and the parameters defined in it).

The Language Environment Runtime Option attribute specifies the program containing the runtime options for the Language Environment. The specified program must be in the DFHRPL concatenation. The default program is DFHAXRO, which is provided as a pre-compiled sample by CICS (see 3.1.3, “Language Environment Runtime Options Program” on page 45 for more details about Language Environment runtime options and the sample programs DFHAXRO).

The Threadlimit attribute specifies the maximum number of concurrent Java applications that can run on the JVM server. The default value is 15. The maximum value is 256. The maximum number of threads that can exist in a CICS region across all JVM servers is 1024. If a JVM server is installed whose threadlimit allocation would cause this limit to be exceeded, CICS will allocate the remaining threads only up to 1024 to that JVM server. If a region has already allocated the maximum number of threads, the JVM server fails to reach the enabled state (see 3.2, “JVM server lifecycle” on page 62 for more details about how to check the number of threads allocated by a JVM server).

See 3.2.1, “JVMSERVER resource definition” on page 63 for more details about the JVMSERVER resource.
Figure 3-3 shows attributes of a JVM server resource definition.

![Attributes of a JVM server resource definition](image)

### 3.1.3 Language Environment Runtime Options Program

The Language Environment Runtime Options file defines options for the Language Environment enclave running a JVM server. The *z/OS Language Environment Programming Guide* documents the Language Environment runtime options. The default values provided by CICS in the DFHAXRO sample program are normally adequate to run JVM servers. However, you might want to change some of the default values. For example, you might want to tune the heap and stack used by Language Environment. In this case, you can either edit the sample assembler source code, which is in the CICSTS42.CICS.SDFHSAMP library or create your own assembler program and compile it. For more information about how to compile the Language Environment Runtime Options Program, see...
the *CICS Transaction Server for z/OS, Version 4 Release 2 information center,*
available at this web page:


For details about how to tune the heap and stack used by the Language Environment, see 5.3, “JVM tuning” on page 136.

Figure 3-4 shows the body of the DFHAXRO source code.

![DFHAXRO sample program](image-url)
3.1.4 JVM server profile for user applications

The JVM server profile defines the runtime configuration of a JVM server.

Figures: Figure 3-5 on page 48, Figure 3-6 on page 50, Figure 3-7 on page 51, and Figure 3-8 on page 52, show the DFHOSGI sample profile provided by CICS.

The parameters that can be defined in a profile, can be divided in four groups:

- **Required parameters** (Figure 3-5 on page 48)
- **Configuration parameters for the JVM server** (Figure 3-6 on page 50 - Figure 3-8 on page 52)
- **Configuration parameters for the JVM** (Figure 3-7 on page 51 and Figure 3-8 on page 52)
- **Environment variables** (Figure 3-8 on page 52)

**Required parameters**

If the two parameters described below are not set correctly, the JVM server will not start:

- **JAVA_HOME**: Must contain the installation directory of the appropriate version of Java.
- **WORK_DIR**: The Java working directory. The JVM server writes the log file, error file, and trace file under this directory. A CICS user must have write permission access to this directory.

Figure 3-5 on page 48 shows a DFHOSGI sample JVM server profile.
Figure 3-5  DFHOSGI sample JVM server profile

```bash
# JVM profile: DFHOSGI
#
# This sample CICS JVM profile is for a JVM server.
#
#***************************************************************************

Symbol Substitution
---------------------

# The following substitutions are supported:
# &APPLID;  => The applid of the CICS region.
# &JVMSERVER;  => The name of the JVMSERVER resource.
# &DATE;  => Date the JVMSERVER is enabled.  Dyyymmd
# &TIME;  => Time the JVMSERVER is enabled.  Thhmmss
#
# Using substitutions means that you can use the same profile
# for multiple regions and still have unique working directories
# and output destinations for each region.
#
# With this substitution
#   ENV_VAR=myvar.&APPLID;.&JVMSERVER;.data
# becomes
#   ENV_VAR=myvar.ABCDEF.JSERVER1.data
# for a JVMSERVER resource with the name JSERVER1 in a CICS region with
# applid ABCDEF.
#
# Note: The continuation character for use with JVMProfiles is '\'.
#***************************************************************************

Required parameters
---------------------

# The set of supported CICS options for JVM servers
# differs from those used with JVM pools.
#
# JAVA_HOME specifies the location of the Java directory.
#
# JAVA_HOME=/usr/lpp/java/J6.0.1_64
#
# Set the current working directory. If this environment variable is
# set, a change to the specified directory is issued before the JVM
# is initialized, and the STDERR, STDOUT and STDERR streams are
# allocated to this directory.
#
# If you do not specify this option, the current working directory is
# left unchanged and the STDERR, STDOUT and STDERR streams are allocated
# to the /tmp directory.
#
WORK_DIR=.
```
Configuration parameters for the JVM server

The parameters explained below are the parameters for the configuration of the JVM server:

- **OSGI_BUNDLES**: Bundles specified on this parameter are installed and made available for the lifecycle of the JVM server. For example, profiles whose JVM servers are configured to run Java Database Connectivity (JDBC) and Structured Query Language for Java (SQLJ) drivers must specify the recommended DB2 Open Services Gateway initiative (OSGi) bundles in the OSGI_BUNDLES options (see 9.4, “Deploying the JDBC and SQLJ samples” on page 301 for more details).

- **OSGI_FRAMEWORK_TIMEOUT**: Specifies the time in seconds after which the JVM server initialization or termination is timed out. It prevents the JVMSERVER resource from getting stuck if start and stop methods of middleware OSGi bundle activators do not return.

- **STDERR, STDOUT, STDIN, and JVMTRACE**: Define the file names where standard error, output and input, and JVM server traces are redirected. If those options are not specified, by default, CICS uses the pattern &APPLID;_&JVMSERVER to prefix the file names.

Figure 3-6 on page 50 shows a DFHOSGI sample JVM server profile.
Specify any directories that contain DLLs required at run time. For example, to use the IBM DB2 Driver for JDBC and SQLJ, add the directory containing the native DLLs to the LIBPATH_SUFFIX option. See the "Application Programming Guide and Reference for Java" relevant to the level of DB2 being used.

LIBPATH_SUFFIX=

*******************************************************************************
# JVM server specific parameters
# -----------------------------
#
# Use the OSGI_BUNDLES option to specify a list of middleware bundles that are installed and activated in the OSGi framework when the JVM is initialized.
The list of bundles must be comma separated. The continuation character is '\'.

#OSGI_BUNDLES=/u/example/pathToBundleDirectory/B1.jar,
#  /u/example/pathToBundleDirectory/B2.jar
#
# This option is used to specify, in seconds, how long the OSGi framework initialization, termination, and middleware bundles initialization are allowed to run before being timed out.
The specified value must be in the range 1-60000. If it falls outside of this range then it will default to 60. If the initialization exceeds the limit, the JVMserver fails to initialize.

#OSGI_FRAMEWORK_TIMEOUT=60
#
*******************************************************************************
# Output redirection
# ------------------
#
# STDOUT, STDERR, STDIN, and JVMTRACE are allocated with file names beginning with &APPLID;.&JVMSERVER;. You can specify different file names using the STDOUT, STDERR, STDIN, and JVMTRACE environment variables.
#
The default file name for JVMTRACE is dfhjvmtc.
# To send the output to somewhere other than a file, specify a user output redirection class. CICS provides a sample that demonstrates this capability. JVMTRACE cannot be redirected.

#USEROUTPUTCLASS=com.ibm.cics.samples.SJMergedStream
*******************************************************************************

Figure 3-6  DFHOSGI sample JVM server profile
Configuration parameters for the JVM

JVM command-line options and system properties can be set in the JVM server profile. The default profile (Figure 3-7) documents the options to set the heap and stack of the JVM and sets the initial heap size to 16 M and maximum heap size to 256 M, and the initial stack size to 128 K. It also documents options to configure the garbage collector. It sets the garbage collector to use the gencon policy and documents the gcthreads option, which specifies the maximum number of helper threads that are used during the garbage collection. Refer to 5.3, “JVM tuning” on page 136 for more details about how to tune the JVM heap and garbage collector.

```
# Configuration parameters for the JVM

### JVM options

- `-Xms` Initial Java heap size, for example `-Xms64M`
- `-Xmx` Maximum Java heap size, for example `-Xmx512M`
- `-Xms0` Initial stack size for native threads (default `-Xms0256KB`)
- `-Xss` Initial stack size for Java threads (default `-Xss128KB`)
- `-Xss` Maximum stack size for Java threads (default `-Xss256KB`)

The following JVM options are a generic set of values.
You can perform workload analysis and provide different values that are more suited to your specific workload.
- `-Xms16K`
- `-Xmx256M`
- `-Xmso28K`

The `-Xgthreads` option sets the maximum number of helper threads allowed for garbage collection. If you do not specify this option, the default is set to (the number of CPUs - 1).
- `-Xgthreads4`

The following option sets the Garbage collection Policy.
- `-Xgcpolicy:gencon`
```

Figure 3-7  DFHOSGI sample JVM server profile
Figure 3-8 shows a DFHOSGI sample JVM server profile.

UNIX System Services environment variables

UNIX System Services environment variables can be used to set some properties for the JVM. The default profile sets the Java dump options through the environment variable JAVA_DUMP_OPTS. The ONANYSIGNAL condition, specified in the value of the variable, instructs the JVM to take Java and system dumps when any operating system signal is received. The JAVA_DUMP_TDUMP_PATTERN environment variable, which is documented in the default profile, can also be set to specify a dump pattern name.
The TZ environment variable can be set to pass to the JVM server and to the JVM the information of the local time zone.

More generally, any parameter that is not a JVM option and that does not start with “-“, is set by CICS as a UNIX System Services environment variable. Therefore, the JVM server profile can be used to set any environment variable that you want to pass to the JVM.

### 3.1.5 JVM server configuration for Java based SOAP pipelines

JVM servers running Java based SOAP pipelines require a particular configuration. The default configuration provided by CICS for the Language Environment run time for JVM servers that run applications is also valid for JVM servers running Java pipelines. The sample program DFHAXRO can be specified in the JVMSERVER definition (see 3.1.3, “Language Environment Runtime Options Program” on page 45 for more details about DFHAXRO).

Yet, the configuration of the JVM server provided by the JVM server profile is different. CICS provides a different sample for Java pipelines JVM servers, DFHJVMAX. Figure 3-9 on page 54, Figure 3-10 on page 55, Figure 3-11 on page 56, and Figure 3-12 on page 57 show the sample profile DFHJVMAX.
Figure 3-9  DFHJVMAX sample JVM server profile
Figure 3-10 shows a DFHJVMAX sample JVM server profile.

```bash
# Specify any directories that contain DLLs required at run time. For
# example, to use the IBM DB2 Driver for JDBC and SQLJ, add the
# directory containing the native DLLs to the LIBPATH_SUFFIX option.
# See the "Application Programming Guide and Reference for Java" relevant
# to the level of DB2 being used.
#
#LIBPATH_SUFFIX=
#
#******************************************************************************
#
# JVM server specific parameters
#                       ---------------------
#
# Use the JAVA_PIPELINE option to configure the JVM Server
# to support Java-based SOAP pipelines.
#
JAVA_PIPELINE=YES
#
#******************************************************************************
#
# Output redirection
#                       -----------
#
# STDOUT, STDERR, STDIN, and JVMTRACE are allocated with file names
# beginning with &APPLID.;JVMSERVER. You can specify different file
# names using the STDOUT, STDERR, STDIN, and JVMTRACE environment
# variables.
#
# The default file name for JVMTRACE is dfhjvmrnc.
# To send the output to somewhere other than a file, specify a user
# output redirection class. CICS provides a sample that demonstrates
# this capability. JVMTRACE cannot be redirected.
#
#USEROUTPUTCLASS=com.ibm.cics.samples.SjMergedStream
#******************************************************************************
```

Figure 3-10  DFHJVMAX sample JVM server profile
Figure 3-11 shows a DFHJVMAX sample JVM server profile.
Figure 3-12 shows a DFHJVMAX sample JVM server profile.

Most of the profile options valid for a JVM server running Java applications are also common to JVM servers running Java pipelines. However, to allow a JVM server to run Java pipelines, the option JAVA_PIPELINE must be set to YES. If JAVA_PIPELINE is not set, the JVM server is configured to run Java applications. The JAVA_PIPELINE option that is set to YES adds Axis2 Java ARchive (JAR) files and CICS Java Pipelines implementation into the CLASSPATH. Java pipelines do not run in an OSGi framework. Therefore, the options OSGI_BUNDLES and OSGI_FRAMEWORK_TIMEOUT are not valid for Axis2 JVM servers.

For all the remaining supported options specified in the sample profile DFHJVMAX, see 3.1.4, “JVM server profile for user applications” on page 47.
## 3.1.6 Quick setup of a JVM server

This section describes the minimum steps that are needed to configure a JVM server in a CICS region. Use the following steps to get a JVM server installed and ready to run Java applications:

1. Edit the CICS JCL file to include a `JVMPROFILEDIR SIT` parameter pointing to a directory where you want to store your JVM server profile (see Figure 3-13). Restart your CICS region. For example, assume that `/u/cics` is the home directory of your CICS UNIX System Services user.

```
SYSDNT=RED6
GRPLIST=(DFHLIST,EPRED6,A0RLIST)
JVMPROFILEDIR=/u/cics/JVMProfiles
USSHOME=/usr/lpp/cicsts/cicsts42
CSDDSN=CICSSEM.EPRED.EPREDC.V42.CSD
```

*Figure 3-13  JVMPROFILEDIR SIT parameter setting in a CICS JCL file*
2. On CICS Explorer, switch to the z/OS perspective. From the z/OS UNIX files window (see Figure 3-14), open the DFHOSGI sample profile under /usr/1pp/cicsts/cicsts42/JVMProfiles. Check that JAVA_HOME is pointing to the installation directory of the 64-bit version of Java 6.0.1. Check that the directory specified by the WORK_DIR option is accessible by the CICS user.

Figure 3-14  JAVA_HOME setting in the JVM server profile
3. Save the profile into the directory specified in the JVMPROFILEDIR.
   a. Click the File menu in CICS Explorer; select Save As.
   b. In the Save as panel (Figure 3-15), specify the destination directory and the name of the profile.

   ![Figure 3-15 CICSJVMS profile being saved in the JVMPROFILEDIR directory](image)

   c. Click OK.

4. Create a JVM server definition using CEDA or CICS Explorer. When using CICS Explorer, take the following steps:
   a. Open the JVM Server Definitions window. From the Definitions menu, select JVM Server Definitions.
b. **Right-click** any area of the JVM Server Definitions window. Select **New**. On the New JVM Server Definition (Figure 3-16), specify the group, the JVMSERVER name, and the name of the profile.

![Figure 3-16](image)

**Figure 3-16** Creation of a JVMSERVER resource called CICSJVMS through CICS Explorer

c. **Click Finish.** The JVM Server Definitions view is updated with the new JVMSERVER resource.
5. Install the JVMSERVER into your region. **Right-click** the JVMSERVER and select **Install**, as shown in Figure 3-17.

![Figure 3-17 Installation of a CICSJVMS JVMSERVER resource](image)

6. In the Perform Operation panel, select the region that you want to install the JVMSERVER to and click **OK**.

7. Check that the status of the JVMSERVER is enabled (see Figure 3-18). On the Operations menu, select **Java → JVM servers**.

![Figure 3-18 Status of CICSJVMS JVM server](image)

The JVM server is now ready to run Java applications. See Chapter 4, “Deploying from CICS Explorer SDK” on page 71 for more details about how to deploy a Java application on a JVM server.

### 3.2 JVM server lifecycle

The JVMSERVER resource follows the lifecycle of CICS resources. It can be defined, installed, enabled, disabled, and discarded. It can be inquired on to check its status and properties. Other than the status, the threadlimit can be changed at run time. To address situations where JVM servers do not respond to
the **disable** command, a command to force the JVM server to quit has been provided (see 3.2.3, “Purging a Java virtual machine server” on page 68).

### 3.2.1 JVMSERVER resource definition

To define a JVMSERVER resource requires some parameters. The Create JVM Server Definition panel in CICS Explorer (Figure 3-19 on page 64) allows the following options to be specified:

- **Resource Group:** An eight-character field that specifies the resource group where the JVMSERVER resource is created. This is a required parameter.
- **Name:** An eight-character field that identifies the JVMSERVER within the group. This is a required parameter.
- **Description:** A 58-character field. This is an optional field.
- **Enabled Status:** The status in which the JVMSERVER is installed. The default value is **Enabled**.
- **Language Environment Runtime Options Program:** Specifies the eight-character program that defines the Language Environment runtime configuration. This is a required parameter. The default is **DFHAXR0** (See 3.1.3, “Language Environment Runtime Options Program” on page 45).
- **JVM profile:** The name of the JVM server profile. This is a required parameter. CICS does not provide a default value. (See 3.1.4, “JVM server profile for user applications” on page 47).
Figure 3-19 shows the Create JVM Server Definition panel.

If the box Open editor is checked (as shown in Figure 3-19), after the JVMSERVER resource is created, the JVM Server Definition Editor (Figure 3-20 on page 65) is opened. Here, the Threadlimit parameter can be set. The threadlimit is the maximum number of concurrent applications that can run simultaneously on a JVM server. The default value for this parameter is 15.
Figure 3-20 shows the Explorer editor of a JVMSERVER resource definition.

![Figure 3-20 Explorer editor of a JVMSERVER resource definition](image)

### 3.2.2 JVMSERVER status transition

The JVMSERVER resource can be installed in the same way as all other CICS resources. However, the installation of a JVMSERVER does not necessarily create a JVM. A JVM is only created when the JVMSERVER is enabled.
Figure 3-21 shows the lifecycle of a JVMSERVER compared to the lifecycle of a JVM.

**Note:** JVMSERVER resources are cataloged by CICS. Therefore, on a warm restart, they are recovered.

![Diagram](image)

**Figure 3-21** Comparison between the JVM server and JVM lifecycles

The JVMSERVER is enabled asynchronously to the **Enable** command, which returns to the user before the JVM server initialization completes. During this phase, the JVMSERVER is set to Enabling status (Figure 3-22).

<table>
<thead>
<tr>
<th>Region</th>
<th>Name</th>
<th>Enable Status</th>
<th>Max Threads</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPRED6</td>
<td>bo</td>
<td>DISABLED</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>EPRED6</td>
<td>CICSVMS</td>
<td>ENABLED</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>EPRED6</td>
<td>DPHVMS</td>
<td>ENABLED</td>
<td>169</td>
<td>0</td>
</tr>
<tr>
<td>EPRED6</td>
<td>JDBCVMS</td>
<td>ENABLED</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>EPRED6</td>
<td>JUSTIN</td>
<td>DISABLED</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>EPRED6</td>
<td>JMSFAIL</td>
<td>ENABLED</td>
<td>255</td>
<td>0</td>
</tr>
<tr>
<td>EPRED6</td>
<td>JMSJDBC</td>
<td>ENABLED</td>
<td>255</td>
<td>0</td>
</tr>
<tr>
<td>EPRED6</td>
<td>RJMSERV</td>
<td>ENABLED</td>
<td>255</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 3-22** JVM server in **ENABLING** state

Similarly, the command to disable the JVMSERVER is asynchronous to the process of disabling a JVMSERVER. During this phase, the JVMSERVER is set
to Disabling status. The JVMSERVER will not move to Disabled status until all the Java applications have terminated. To force the JVM server to stop without letting applications quiesce, the Purge, Forcepurge, and Kill commands can be used (See 3.2.3, “Purging a Java virtual machine server” on page 68).

When a JVMSERVER is enabled, CICS creates a Language Environment enclave and starts the JVM in the Language Environment enclave. For an OSGi-enabled JVM server, CICS initializes the OSGi framework and installs a set of system bundles, and any middleware bundles that have been specified in the OSGI_BUNDLES option of the JVM server profile (Figure 3-23).

![Figure 3-23 JVM server enablement](image)

The OSGi system bundles are the Equinox implementation of the OSGi framework. The CICS system bundles are the implementation of OSGi in CICS. The middleware bundles are third-party bundles that are needed by the user applications, such as the JDBC driver bundle. If any of these components fail to activate, the JVM server fails to become enabled.
JVM task control blocks
CICS uses a new type of task control block (TCB), T8 TCBs, to run Java applications in a JVM server. These TCBs are both CICS and Java enabled and are limited by the THREADLIMIT attribute in the JVMSERVER resource. The JVM server operates a queuing mechanism if it has reached its threadlimit, and tasks are suspended waiting on a JVMTHRD. As other tasks finish, the waiting tasks are then allocated to the freed T8 TCBs. In addition to this, each JVM server also reserves a dedicated system management T8 TCB. This is used to run SPI commands. This ensures that system management operations do not block even when the JVM server is running at maximum capacity.

3.2.3 Purging a Java virtual machine server

During a JVM server disable, all the installed bundles are stopped, activator stop methods are run, and the OSGi framework is terminated. Lastly, the JVM is terminated. When the JVMSERVER reaches the disabled state, it can be discarded.

When a JVM server does not respond to the disable command, it is most likely because an application is misbehaving or not returning in a timely fashion. In this scenario, there are different options available to force execution of applications to stop, and the JVM server to reach the disabled state.

To quiesce a JVM server, you can use the JVMSERVER SET command, via the CEMT command or CICS Explorer to either purge, forcepurge, or kill the JVM server and the tasks running within it.

For each of these forced quiesce commands, a two-phase termination is carried out on all tasks running in that JVM server. One phase halts the task if it is running Java code. The second phase halts the task if it is running CICS code (as a result of a JCICS command). When a thread is halted within Java code, it might leave the JVM inconsistent or unstable. For this reason, the forced quiesce
commands will always result in the JVM being terminated and recycled. A CICS task purge alone (if it is even successful in terminating the task) will not cause the JVM to be recycled and is therefore far more dangerous to use than the dedicated \texttt{JVMSERVER quiesce} commands.

\textbf{Purge commands}

If you specify the \texttt{PURGE} command, tasks running in the JVM are halted at the Java level. Tasks running in CICS are issued with a TASK PURGE. Tasks are purged by CICS only if system and data integrity can be preserved and if PURGE is allowed. Otherwise, the JVMSERVER stays in Disabling state.

The \texttt{FORCEPURGE} type can be issued only after a \texttt{PURGE}. As with \texttt{PURGE}, any tasks that are running Java code are halted at the JVM level. Any tasks running CICS code are issued with a TASK FORCEPURGE. Data integrity is not guaranteed.

The \texttt{KILL} type of purge performs a JVM thread level halt, and terminates any tasks running on the JVM server by issuing the \texttt{SET TASK KILL} command. The integrity of the system is not guaranteed and CICS might terminate abnormally. If you use \texttt{FORCEPURGE} or \texttt{KILL}, you are advised to restart CICS.
Deploying from CICS Explorer SDK

In this chapter, we describe the deployment process for getting a Java application written on the workstation to be run in CICS. We use the Eclipse integrated development environment (IDE), enhanced with the CICS Explorer software development kit (SDK) plug-in, to demonstrate the deployment process through the various stages up to execution in a stand-alone CICS region or a CICSPlex environment.

This chapter does not focus on the development process. Instead, it focuses on the process necessary to deploy and test an existing application. Java applications can be tested in the IDE to a point, but when you add CICS specific Java class library for CICS (JCICS) application programming interface (API) commands, testing must be performed in a CICS region or in a Rational Developer for System z Unit Test (RD/z UT) environment.

Systems Programmers can use the information in this chapter to get started with Java in their CICS environment before turning it over to the developers to create applications. We use the CICS provided samples to demonstrate the deployment process.
4.1 The CICS Explorer development platform

For Java applications intended to run in a CICS Transaction Server (TS) environment, one key piece of software is required, the CICS Explorer SDK. When installed in an Eclipse-based Java development environment, the CICS Explorer SDK allows you to create Java application components, which are known as Open Services Gateway initiative (OSGi) bundles (or plug-in projects). In turn, these OSGi bundles are packaged into a CICS managed resource called a CICS bundle project. The CICS bundle project can be installed into a CICS TS V4.2 region and the Java applications contained within the CICS bundle can be ran as OSGi services within the OSGi framework.

You can use any platform or IDE that will generate OSGi bundles to write your Java application. However, the deployment step must use the CICS Explorer SDK to package and deploy your OSGi bundles into a CICS bundle project and up to a zSeries file system (zFS). The CICS Explorer SDK is installed into an Eclipse-based Java IDE as a plug-in. For more information about downloading and installing the CICS Explorer SDK into an Eclipse IDE, see Appendix A, “Installing the CICS Explorer software development kit” on page 367.

In this chapter, we also use the CICS Explorer rich client platform (RCP) alongside the CICS Explorer SDK. This allows us to perform comparisons on various zFS file structures.

**Note:** You might wonder why you should use the CICS Explorer at all. Why not just use the Eclipse SDK all the time? It is a convenience issue. If you never work with Java code, just use the CICS Explorer. If you work with Java programs in CICS, you *must* use the Eclipse IDE with the CICS Explorer SDK plug-in. In our case, we find that we can run both at the same time and have extra windows for comparison purposes.

**JVM pool versus JVM server**

In CICS TS V4.2, there are two ways you can run Java applications. In this chapter, we describe using the JVM server run time only. Chapter 10, “Migration” on page 315 covers migration to a JVM server from the JVM pool environment.
4.2 The deployment process

The deployment process is the steps required to take an OSGi Java application package from the development stage on a workstation, all the way to the execution of the application in a target CICS region or cluster of regions.

4.2.1 Key steps of the deployment process

Here is an outline of the development and deployment process that is followed in this chapter:

1. Develop Java OSGi plug-in project components on the workstation, using an Eclipse IDE.
2. Package Java plug-in projects into a CICS bundle project using the CICS Explorer SDK.
3. Deploy the CICS bundle project to a zFS directory on z/OS.
   From UNIX System Services, the application can be installed into a target CICS region or into a CICSPlex group of application-owning regions (AORs).
   To achieve this, the Systems Administrator must define the following components:
4. A JVM server and a CICS bundle definition.
5. The application resources to the CICS region, such as transaction and program definitions.
The benefit of using the CICS examples is that when they are installed, they can be used like installation verification procedures (IVPs) to verify that the systems are running correctly. They can also provide a good reference implementation that you can extend and customize to create your own Java applications.

We deploy into two different environments, a stand-alone single region, and a CICSPlex environment consisting of multiple regions.

### 4.2.2 Source code management and repositories

In a large enterprise environment, you will most likely have source code management products and repositories. Although not used in our test environment, source code management can easily be worked into the process.
Figure 4-2 shows the additional step of storing your Eclipse projects to a source code repository or management product and then deploying via the CICS Explorer SDK. The addition of a source code management system provides the change control and management services required with a modern development process.

![Diagram of integration with a source code management product]

**Figure 4-2   Integration with a Source Code Management Product**

### 4.3 Deploying into a stand-alone CICS region

In this section, we use the CICS Explorer SDK to create the CICS Java examples. The examples are packaged in a CICS bundle project and deployed to a stand-alone CICS region called *EPREDB*. Within this region, we define a JVM server, a CICS bundle referencing the deployed application, and the transaction and program definitions required to run the example applications.

The Systems Administrators usually provide a developer with the system information that they will use, such as the region’s name, APPLID, and target DFHRPL load library. For Java applications, you need two more pieces of information:

- The JVM server name.
- The zFS location (bundle directory) to which the CICS bundle project, or application, should be deployed.
4.3.1 Single region setup review

Our target environment is a stand-alone CICS region called EPREDB. The user ID that is associated with EPREDB is CICSRS5, and our region’s resources are in zFS directory /u/cicsrs5. Figure 4-3 shows the expanded zFS directory structure. Under this structure, we have a zFS subdirectory identified by the application identifier (APPLID) of our target CICS region (EPREDB). This region directory contains subdirectories for the Java applications, the JVM profiles, and a work directory.

Application directory
JavaApps is the zFS directory that is created to contain all our CICS bundle projects deployed to region EPREDB (Figure 4-4.). This is the directory into which all our CICS bundle projects were exported from the Explorer SDK. The subdirectories correlate with individual CICS bundle definitions.
JVM profiles directory
The JVM profiles directory is referenced via the SIT parameter JVMPROFILEDIR. It tells CICS where to find the JVM profiles used in that region. Figure 4-5 shows the value of that SIT parameter for our CICS region EPREDB.

```
JVMPROFILEDIR=/u/cicsrs5/EPREDB/JVMPfiles
```

*Figure 4-5  SIT parameter for stand-alone region EPREDB*

Expanding the JVMProfile directory (Figure 4-6), you can see the profiles used by EPREDB. Copy the sample profiles from their shipped location into your JVMPROFILEDIR location. Then, customize the samples for your specific environment. Notice that we did not copy profiles DFHJVMPR and DFHJVMCD because these are JVM pools examples and *not* for use in the JVM server.

*Figure 4-6  Expanded JVMProfile directory*

DFHJVMAX is provided for Axis2 web services that run in a JVM server (not covered in this chapter). The profile, DFHOSGI, is used for JVM server-based Java applications.
Work directory
In the JVM profile, you set the WORK_DIR parameter to tell CICS in which zFS directory to store trace, output, and error files. Figure 4-7 shows our expanded WORK_DIR structure.

The following list describes the JVM server output files:

- dfhjvmerr: File containing JVM error messages from stderr
- dfhjvmin: File used as stream input from stdin
- dfhjvmout: File containing stream output from stdout
- dfhjvmtrc: File containing JVM server trace output

The EPREDB subdirectory in the WORK_DIR directory is used by the OSGi framework and contains a subdirectory for the JVM server, RBJVMSRV. This subdirectory contains cache information used by the OSGi framework along with the OSGi log files. You can find more information about these in the CICS TS V4.2 Java Applications in CICS manual, available at this link:


Tip: By default, CICS prefixes the JVM server output files with the APPLID and the JVM server name. However, you can override the location and file name patterns for these output files via options in the JVM profile. See the comments in the sample JVM profiles for the symbolic substitution variables and some examples.

4.3.2 Example Java class library for CICS application

The Java example applications that ship with CICS Explorer SDK can be used to demonstrate the deployment process.
Creating the example projects
Create the example OSGi plug-in projects and CICS bundle projects with the following procedure.

1. Right-click in the Package Explorer view. Select New → Example to open the example selection wizard (Figure 4-8).

![Figure 4-8  Eclipse SDK: Importing the example OSGi plug-in projects](image-url)
2. Select **CICS Hello Examples**, as shown in Figure 4-9. Continue by clicking **Next**, and then click **Finish**.

![Figure 4-9 Selection to import CICS Hello Examples](image)

3. Repeat these steps starting from Figure 4-8 on page 79, selecting **CICS API Examples** and **CICS Web Example**.

**Note:** For the examples, we used the default project names. However, you are free to rename projects if it is done consistently throughout the following procedures.
You now have three OSGi plug-in projects in your Package Explorer view, as shown in Figure 4-10.

Figure 4-10  Eclipse SDK with three OSGi plug-in projects

You can expand the projects and review the code if you want. To deploy the applications to the z/OS environment, we must package them into a CICS bundle project. The CICS bundle project is then deployed to the bundle directory referenced by the CICS bundle definition in CICS.

To create the example CICS bundle project, perform the following actions:

1. Starting from the Package Explorer view, in Figure 4-11, right-click in the Package Explorer view. Select New \(\rightarrow\) Example.
2. From the CICS Java folder, select **CICS Application Bundle Example** (Figure 4-12).

![Figure 4-12 Create a CICS example bundle project](image)

3. Give your project a name, accept the defaults, and click **Finish**.
   
   Your Package Explorer view should now contain three OSGi plug-in projects and one CICS bundle project.
Figure 4-13 shows the CICS bundle project with the three OSGi plug-in project .osgibundle entries that point to their associated OSGi plug-in project components. The OSGi plug-in projects are the applications and the CICS bundle project is a wrapper to facilitate the deployment into CICS.

Figure 4-13   Eclipse SDK with three OSGi plug-in projects and one CICS bundle project

**Setting the target Java virtual machine server**

One of the key pieces of information that a developer requires is the JVM server name. In this section, we set the target JVM server for the example applications.

The examples are preset to run in a JVM server called *DFHSJVMS*. For demonstration purposes, we change the target JVM server so that we can deploy into a different JVM server. Our target CICS region, EPREDB, is configured to use JVM server, RBJVMSRV. This value is specified in the applications .osgibundle resource file.
In the Package Explorer view, double-click the **hello.osgi** file to open the editor and replace DFH$JVMS with RBJVMSRV (Figure 4-14).

Save and close the editor and repeat for the other two .osgbundle files, jcics.osgi and web.osgi.

### 4.3.3 Deploying to z/OS UNIX System Services

The CICS bundle project is exported to z/OS UNIX System Services and installed in a target CICS region. The deployment step uses the CICS Explorer SDK to package up the OSGi plug-in projects using the information from the .osgbundle resource files to create a Java ARchive (JAR) file for each OSGi plug-in project. There is also a cics.xml manifest file used to describe the CICS bundle and its contents.

**Note:** The bundle project, when exported to z/OS UNIX System Services, is not able to be recreated from the exported data on the z/OS side. Therefore, you must save your bundle project and store it for future use in a suitable location or source code repository.

The second key piece of information that a developer might require is the zFS directory that the application should be deployed to. Figure 4-4 on page 76 shows the zFS directory structure for the EPREDB region. CICS will be configured to install the example applications from /u/cicsrs5/EPREDB/JavaApps. As described in 4.2, “The deployment process” on page 73, you might be
required to deploy into a staging zFS directory or a Source Code Management repository. The System Administrator would copy from there into the appropriate CICS target zFS directory.

**Deploy the CICS bundle project**

From the Package Explorer view (see Figure 4-15), right-click the CICS bundle project and select **Export Bundle Project to z/OS UNIX File System**... If you are not already connected to z/OS, you are prompted to log on.

The export to the UNIX z/OS file system selection wizard displays, allowing you to select the target destination for your CICS bundle project. Using the Parent Directory selection field, enter the target zFS directory where you want to deploy the CICS bundle project to. Notice as you type, the Bundle Directory field is automatically completed with the zFS directory name concatenated with the CICS bundle project name.

**Note:** The CICS Explorer SDK saves the target zFS directory information and recalls it for future exports of the same CICS bundle project.
Figure 4-16 shows deploying the CICS bundle project, com.ibm.com.cics.server.examples into zFS directory /u/cicsrs5/EPREDB/JavaApps.
Click **Finish** and the CICS Explorer SDK starts the copy operation. If there are errors, you receive a failure response with an appropriate message, as shown in Figure 4-17. In this example, the CICS bundle project already exists and you must check the **Clear existing contents of Bundle directory** box to allow the new copy to overlay the existing copy.

![Figure 4-17 Export to z/OS UNIX File System: Failure](image)

*Figure 4-17  Export to z/OS UNIX File System: Failure*
The copy operation gives a status indication bar, as shown in Figure 4-18.

Figure 4-18  Export to z/OS UNIX File System: Progress status
When complete, the Export to z/OS UNIX File System dialog window closes. Figure 4-19 shows the completion status message on the bottom of the window. If you open the console view, you see a breakdown of the files that were copied to the zFS file system.

Exploring the console message from the Console view in Figure 4-19, you can see that it created a mirror of the CICS bundle project on zFS along with the three JAR files, one for each OSGi plug-in project.
Saving your bundle definition

If you want to back up your CICS bundle project, you can export the CICS bundle project to your local file system for safe keeping or optionally store it in a Source Code Management repository. Follow these steps:

1. **Right-click** your CICS bundle project and select **Export** (Figure 4-20).

![Figure 4-20 Export a bundle project](image)
2. Select **Archive File**, as shown in Figure 4-21, and click **Next**.

*Figure 4-21  Export as a compressed file*
3. In the selection boxes, check all the resources that make up your CICS bundle project, as shown in Figure 4-22.

![Figure 4-22 Export to a compressed file selection](image)

This saves your CICS bundle project to your local workstation as a compressed file. From there, you can save it in an external location or repository.

For more information about the backup and restore process, see 4.5, “Exporting and reimporting your project” on page 118.

### 4.3.4 Installing into the target CICS region

Now that our application is on z/OS, we can install it into a target CICS region. To accomplish this, we need a few resources. As with any CICS application, you need some standard CICS resource definitions such as Transaction and Program definitions. For Java applications in CICS TS V4.2, you also need a Bundle definition and a JVM server definition.
JVM server definition

The example Java programs must run in a JVM server; therefore, a CICS JVM server definition is required. You can think of a JVM server as a CICS region-based resource, perhaps similar to a TCP/IP service definition where you define ports for the region, then let applications use them. The same is true for the JVM server; define one or more to the region and let applications use them. Similar to other region-based resources, you might want to segregate them or create one and let everyone share it.

In our examples, we are using a shared model, one JVM server per CICS region. Thus, the JVM server definition would be part of the base CICS region resources, installed early in the startup cycle.

Shown in Figure 4-23 is the JVM server definition. The right side of the panel, which shows the definition attributes, is open for editing.

Figure 4-23  JVM server definition for CICS region EPREDB
Figure 4-24 shows the installing group, JEXMPJVM, from the Resource Group Definitions view and the selection of the EPREDB target region.

This selection installs and activates the single shared JVM server called \textit{RBJVMSRV}, in region EPREDB. Chapter 3, “Setting up a Java virtual machine server” on page 41 goes into detail about the setup of a JVM server. One aspect of configuration that we explore further is the JVM profile. The JVM profile defines the CICS region's working directory, the location of Java on z/OS, the JVM's garbage collection policy, heap storage settings, dump options, shared class cache, and many more configuration settings. The profiles used in these samples are fairly simple. For complete details about the setup and options available in the JVM profiles, see the CICS TS V4.2 Information Center at the following web page:

\url{http://pic.dhe.ibm.com/infocenter/cicsts/v4r2/index.jsp}

**CICS application resources**

CICS provides a sample resource definition online (RDO) group, \textit{DFH$OSGI}. However, we are changing the JVM server name. Therefore, we copied the sample definitions to a new resource group called \textit{JEXMPAPP}. 

Figure 4-25 shows a screen capture from the CICS Explorer showing the transaction definitions for the Java Example Application.

The program definitions are shown in Figure 4-26 and note the resource group, JEXMPAPP, where they all belong.

Figure 4-26 shows the program definition for DFJ$JHE2, in the editor. Clicking the Java tab, highlighted in red, shows the Java related attributes. As mentioned previously, we changed the JVM server value from the default value of DFH$JVMS to RBJVMSRV.
Figure 4-27 is the same program definition displayed via CEDA for comparison.

Both the CEDA program definition and the CICS Explorer program definition have the value examples.hello.HelloCICSWorld set as the JVMClass. This is the fully qualified Java class name, which contains the main method to be run. More accurately for an OSGi application, it is the value of the CICS-MainClass manifest entry specified in the target OSGi bundle.
Figure 4-28 shows the installation group, JEXMPAPP, from the Resource Group Definitions view, and the selection of the EPREDB target region. This selection installs and activates the sample transaction and program resources in the CICS region, EPREDB.

We now have a JVM server defined, installed, and ready for use. The application resources are installed and available. All that is left to do is install the CICS bundle definition.

**Bundle definition**

A CICS bundle definition contains a reference to the zFS directory that contains our CICS bundle project. See 4.3.3, “Deploying to z/OS UNIX System Services” on page 84.
Shown in Figure 4-29 is the bundle definition. The right side of the panel, which shows the definition attributes, is open for editing.

Figure 4-29 Bundle Definitions panel for the Java Example Application

This selection installs and activates the CICS bundle definition in region EPREDB.

Figure 4-30 CICS Explorer installation of the bundle resource
If you install everything as a single group through CEDA, you get an initial bundle installation failure message. However, the bundle installs correctly after a short delay. No such error message is displayed if you install the group through CICS Explorer. You can see the message flow in Example 4-1.

If you have worked with bundles before, you know that the bundle controls the lifecycle of all the resources defined in it. This is also true for OSGi applications. If you disable the bundle, the Java application is disabled. Disabling the bundle does not disable the JVMSERVER, which is a resource in its own right, managing its own lifecycle.

In Example 4-1, we created a test group that is called TESTIT, which contained the transaction and program definitions along with the JVM server and bundle definitions.

Example 4-1  Bundle installation lifecycle management

DFHPG0101 04/11/2012 04:55:08 EPREDB TCP66024 CICSUSER CEDA Resource definition for DFJ$JHE1 has been added.
DFHRD0101 04/11/2012 04:55:08 EPREDB TCP66024 CICSUSER CEDA INSTALL PROGRAM(DFJ$JHE1)
DFHPG0101 04/11/2012 04:55:08 EPREDB TCP66024 CICSUSER CEDA Resource definition for DFJ$JHE2 has been added.
DFHRD0101 04/11/2012 04:55:08 EPREDB TCP66024 CICSUSER CEDA INSTALL PROGRAM(DFJ$JHE2)
DFHXM0101 04/11/2012 04:55:08 EPREDB TCP66024 CICSUSER CEDA TRANSACTION definition entry for JHE1 has been added.
DFHRD0104 04/11/2012 04:55:08 EPREDB TCP66024 CICSUSER CEDA INSTALL TRANSACTION(JHE1)
DFHXM0101 04/11/2012 04:55:08 EPREDB TCP66024 CICSUSER CEDA TRANSACTION definition entry for JHE2 has been added.
DFHRD0104 04/11/2012 04:55:08 EPREDB TCP66024 CICSUSER CEDA INSTALL TRANSACTION(JHE2)
DFHSJ0910 04/11/2012 04:55:08 EPREDB CICSUSER JVMSERVER RBJVMSRV has been created.
DFHRD0131 I 04/11/2012 04:55:08 EPREDB CICSUSER JVMSERVER RBJVMSRV
DFHRL0107 I 04/11/2012 04:55:08 EPREDB CICSUSER The CICS resource life-cycle manager has started to create the BUNDLE resource JEXMPBUN.
The OSGi bundle has not been installed because the JVM server RBJVMSRV is not enabled. OSGi bundle symbolic name com.ibm.cics.server.examples.hello, version 1.0.0.

The OSGi bundle has not been installed because the JVM server RBJVMSRV is not enabled. OSGi bundle symbolic name com.ibm.cics.server.examples.jcics, version 1.0.0.

The OSGi bundle has not been installed because the JVM server RBJVMSRV is not enabled. OSGi bundle symbolic name com.ibm.cics.server.examples.web, version 1.0.0.

Example 4-1 on page 99 highlights some of the interesting messages in bold/italic. You can see that the bundle installation fails but is later corrected.

Using the CICS Explorer to install the group, TESTIT, generates the same message flow, but the final installation message is successful.

### 4.3.5 Testing the sample application

Here, we test the JHE1 and JHE2 transactions, which are the Hello World examples that are designed to run from a 3270 terminal.
We use the CICS Explorer to display all the resources used by this test. Figure 4-31 shows the successful execution of the JHE2 transaction from the single stand-alone region EPREDB. Below that are some runtime statistics generated by running several hundred JHE1 and JHE2 transactions through EPREDB.

![Figure 4-31 Single-region test of JHE2 from EPREDB](image)
Figure 4-32 shows transaction information for JHE1 and JHE2. Notice that JHE1 is actually two programs: an initial C language program that then calls the Java class, DFJ$JHE1.

![Figure 4-32 Testing: Transactions view](image1)

Figure 4-33 shows the two HelloWorld program definitions in CICS Explorer. We customized the columns to show Java related fields such as the JVM class and JVM server.

![Figure 4-33 Testing: Programs view](image2)
Figure 4-34 shows information about the JVM server, RBJVMSRV.

The display shows the associated JVM profile and the JVM garbage collection policy and use count. The contents of the DFHOSGI profile are shown in Figure 4-40 on page 107.

Figure 4-35 shows the bundles installed in region EPREDB. The directory information allows you to tie the region to the applications that it manages.
Figure 4-36 shows the components of the bundle.

Figure 4-36  Testing: Bundle Parts view

Figure 4-37 shows the relationship of the OSGi bundles to the CICS bundle.

Figure 4-37  Testing: OSGi Bundles view
Figure 4-38 shows the OSGi Services view and the bundle that they are associated with.
Figure 4-39 is the SIT Parameters (COMBINED) view, which is used by the EPREDB region. It shows the **USS Home** and **JVM Profile Directory** parameters.

![Figure 4-39 Testing: SIT Parameters view](image)

The two JVM class cache parameters that are listed under *Java* and are not in red, are not used by EPREDB. They are only relevant for JVM Pools.
Figure 4-40 shows the JVM profile that is used by RBJVMSRV.

![Image of CICS Explorer SDK](image)

**Figure 4-40** Testing: JVM profile, DFHOsGI

### 4.4 Deploying into a CICSPlex

In this section, we deploy the same application as in 4.3, “Deploying into a stand-alone CICS region” on page 75. However, instead of a single region deployment, the application runs in a multi-region CICSPlex.
The EPRED CICSPlex is made up of one TOR (EPREDT), nine AORs (EPREDn), and one file-owning region (FOR) (EPREDF) along with a CICSPlex SM address space (CMAS) (EPREDC) and a WUI server (EPREDW), as shown in Figure 4-41.

4.4.1 Multi-region setup review

Our target environment is a group of cloned AOR CICS regions, called EPRED1-EPRED9. There are no files in this example, but the FOR or DB2 would provide the data access and all routable work enters the CICSPlex through the Transport Owning Region, EPREDT.

The user ID that is associated with the EPRED CICSPlex is CICSRS1. All regions are running under the same user ID, and our CICS resources are in a common zFS directory /u/cics/. Figure 4-3 on page 76 shows the expanded zFS directory structure, in which we have a shared zFS directory tree for our target AORs, with subdirectories for deployed bundles (JavaApplications), JVM profiles (JVMProfiles), and output files (workdir). See Figure 4-42.
Application directory

/u/cics/JavaApplications is a shared zFS directory that stores the deployed Java applications (Figure 4-43). The CICS regions locate this directory via a shared bundle definition, which is installed into all the AORs.

![Figure 4-43  Java applications zFS directory](image)

JVMProfiles directory

/u/cics/JVMProfiles is the location where all the JVM server profiles are stored. This is set via a common SIT parameter called JVMPROFILEDIR. Figure 4-44 shows the SIT parameter that is used in our AORs, which were set in all the CICS regions.

```
JVMPROFILEDIR=/u/cics/JVMProfiles
```

![Figure 4-44  SIT parameter for CICSPlex regions](image)
Expanding the JVMProfile directory, see Figure 4-45. You can see the profiles used by the JVM servers in EPRED1 - 9 CICS regions.

![JVMProfile Directory](Image)

Figure 4-45 Shared JVMProfile directory

Profile CPLXOSGI is the common shared profile that is used for our sample Java application.

**Work directory**

/u/cics/workdir is the work directory for the JVM servers. This is set in the JVM profile, via the **WORK_DIR** parameter. It is used for storing JVM server output files and the OSGi cache. Figure 4-46 shows the work directory and **Dump Pattern** parameters from the JVM profile, CPLXJVMS, which is used to create the shared work directory structure shown in Figure 4-7 on page 78.

| WORK_DIR=/u/cics/workdir/&APPLID; | JAVA_DUMP_TDUMP_PATTERN=DUMP.&APPLID;&JVMSERVER;&DATE;&TIME; |

Figure 4-46 JVM profile parameters used by the CICSPlex regions

The &APPLID, &JVMSERVER, &DATE, and &TIME variables are used to create unique resource files for the region output and dumps, which are required when you share a common directory as in a CICSPlex.
Because this is a CICSPlex environment, the structure is flexible and allows sharing across all the regions. Figure 4-47 shows the shared work directory structure. Each region has its own subdirectory so that each region’s JVM server output files can be easily located in their own unique directory.

Figure 4-47  Shared JVM server subdirectory

Figure 4-47 also shows the EPRED9 subdirectory in the workdir directory, which is associated with the JVM server, CPLXJVMS, and is used by the OSGi framework to save diagnostic information related to the OSGi cache and OSGi logs. If you expand the other directories, you would see that all the other EPRDx regions have the same JVM server subdirectory.

4.4.2 Application test and component overview via CICS Explorer

We are testing only the JHE1 and JHE2 transactions, which are the Hello World examples, and are designed to run from a 3270 terminal. As this section suggests, this is a multi-region test using CICSPlex regions EPRED1 - 9 along with EPREDT as the point of entry, which will then work load balance all incoming traffic down into the AORs using CICSPlex System Manager’s Workload Manager (WLM) specifications.
We use CICS Explorer to display all the resources used by this test. Figure 4-48 shows the successful execution of the JHE2 transaction from the terminal-owning region (TOR), EPREDT. The actual transaction runs in one of the AORs, EPRED1 - 9. Below that, there are some displays that show runtime statistics after running several hundred JHE1 and JHE2 transactions through EPREDT.
Figure 4-49 is the SM Administration view showing the routing region, EPREDT, in the top section. The middle and bottom sections are the Workload Targets and Workload Target Status views showing the status and health of the AOR regions, EPRED1 - 9.
Figure 4-50 shows transaction information for JHE1 and JHE2. Notice that JHE1 is made up of two programs, an initial C language program, DFH$JSAM that calls the Java class, DFJ$JHE1.

![Figure 4-50 Testing: Transactions view](image)

Figure 4-50 shows transaction information for JHE1 and JHE2. Notice that JHE1 is made up of two programs, an initial C language program, DFH$JSAM that calls the Java class, DFJ$JHE1.

Figure 4-51 shows the Programs view for the two HelloWorld programs. We customized the columns to show Java related fields such as the JVM class and JVM server. This allows you to identify what JVM class is associated with each program.

![Figure 4-51 Testing: Programs view](image)

Figure 4-51 shows the Programs view for the two HelloWorld programs. We customized the columns to show Java related fields such as the JVM class and JVM server. This allows you to identify what JVM class is associated with each program.
Figure 4-52 shows information about the JVM server, CPLXJVMS. As shown, all regions in the CICSPlex are running the same JVM server.

Notice that the JVM Servers view shows you the JVM profile associated with each JVM server and the garbage collection policy and use count. The contents of the CPLXOSGI profile are also displayed in Figure 4-57 on page 118.

Figure 4-53 shows the bundles installed in regions EPRED1 - 9. From this display, you can tie the bundle with its source input directory.

**Important:** To make a Java program threadsafe does not just mean that the CONCURRENCY parameter is properly set, it means that the application is coded to threadsafe standards. See 10.6, “Thread-safe Java applications” on page 365.
Figure 4-54 shows the components inside each bundle. The Enable Status column is useful to help identify the status of resources in a bundle.

![Image: Testing: Bundle Parts view](image1)

Figure 4-54  Testing: Bundle Parts view

Figure 4-55 shows the OSGi Services available and the bundle they are associated with.

![Image: Testing: OSGi Services view](image2)

Figure 4-55  Testing: OSGi Services view
Figure 4-56 shows the SIT Parameters (COMBINED) display giving the **USS Home** and **JVM Profile Directory** attributes.

![SIT Parameters Display]

All EPREDx regions are sharing SIT parameters. If you display the SIT parameters for each region, they are identical where applicable.
Figure 4-57 shows the shared JVM profile that is used by CPLXJVMS.

![Image of CICS Explorer showing JVM profile](image)

Figure 4-57   Testing: JVM profile, DFHOSGI

### 4.5 Exporting and reimporting your project

You are advised to regularly back up your projects. Additionally, you might want to share your project with other team members. Because it exists initially on your workstation, you must store a copy on a server or in a source code management product so others can gain access to it.
4.5.1 Creating an archive file for backup or storage

From the Package Explorer, right-click your CICS bundle project and select Export, as shown in Figure 4-58.

Figure 4-58  Export a bundle project
Select **Archive File**, as shown in Figure 4-59, and click **Next**.
In the selection boxes, check all the resources that make up your CICS bundle project. See Figure 4-60.

![Figure 4-60 Export to compressed file selection](image)

This saves your CICS bundle project to your local workstation as a compressed file. In this example, the archive file was saved to the desktop. From there, it can be saved, copied, or moved to an external location or repository.

In this scenario, we are backing up and restoring only the CICS bundle project portion of our project. A complete project is made up of one or more OSGi bundle projects and the associated CICS bundle project. You can follow the same steps to back up the OSGi portion of the project. And, you can combine them together or leave them separate for maximum flexibility. The restore steps are also the same.

### 4.5.2 Restoring from an archive file

The previous section shows you how to make an archive file for backup and storage. However, you cannot restore from the archive file directly and re-create the original project structure.
To recover a project from an archive file, all you must do is extract the archive file into your target workspace and then import in the Explorer SDK. Proceed with the following instructions:

1. Using the Package Explorer view, select **Import → Existing Projects into Workspace**. Extract the archive file.

2. Identify your workspace directory from the menu bar. Select **File → Switch Workspace → Other**. The workspace listed in the drop-down box, as shown in blue in Figure 4-61, is your current workspace.

![Workspace Launcher](image)

*Figure 4-61  Finding your workspace directory*

3. Using your Windows File Explorer, locate your archive file and right-click it. Select **Extract All**.
Figure 4-62 shows the current contents of the workspace currently used by our Eclipse IDE with a CICS Explorer SDK plug-in before restoring the MyTestCICSBndlToExportandImport bundle project.

4. Again, locate the saved bundle project on the workstation. Right-click it and select **Extract All** to unpackage the compressed file into the current workspace.

5. Clicking **Extract** expands the compressed file to the workspace. Figure 4-63 shows a screen capture of our workspace directory showing the restored bundle project, MyTestCICSBndlToExportandImport.
6. In the Project Explorer view, right-click and select **Import**. Select **Existing Projects into Workspace** (Figure 4-64). Notice the steps previously taken put this project into the workspace so it can be imported as a previously existing project.

![Import existing projects from the current workspace](image)

*Figure 4-64  Import existing projects from the current workspace*
7. From the Import window, select the **Browse** icon on the upper right and locate the project, `MyTestCICSBundleToExportAndImport`, that we just restored back into the workspace, as shown in Figure 4-65.

![Figure 4-65  Browse and select project to import](image)
8. With the import operation complete and back at the Project Explorer, you can see the newly restored bundle project that is associated with the examples.hello package, which is shown in Figure 4-66.

![Figure 4-66](image_url)  Package Explorer after restoring the missing CICS bundle project

### 4.6 Deployment best practices

This section is a description of best practices regarding setup and deployment of your Java applications and resources.

#### 4.6.1 UNIX System Services directories

Java applications are not the only components that reference directories in UNIX System Services. Therefore, you might want to create a structure that allows you to organize your Java resources along with other zFS artifacts such as Web Services, Event Processing, and Atom Feeds.

If you have a CICSPlex, ensure that the resources are shareable across all CICS regions in your CICSPlex. To this end, you should ensure that you have a well-designed directory structure and the correct read and write permissions for all CICS regions in your CICSPlex. In addition, if your CICSPlex spans multiple logical partitions (LPARs), you should ensure that you have a sysplex shared zFS filing system to facilitate sharing of files between CICS regions.
4.6.2 JVM profiles

Do not use the default zFS location that CICS installs the sample JVM profiles into. If you do, any future maintenance might overlay them with new updated profiles and you lose your settings. Copy the profiles that you need to a new location related to each target region or your target CICSPlex. You can change the names of the profiles to something more specific to your application or product.

4.6.3 OSGi bundle dependencies

Ensure that all OSGi bundles that comprise a Java application are deployed in the same CICS bundle. Using this method, you can manage the application as a single entity by using the bundle resource. If there are dependencies between OSGi bundles, deploy them in the same CICS bundle. When you enable the CICS bundle resource, CICS ensures that all the dependencies between the OSGi bundles are resolved.

If you have multiple applications that have dependencies on an OSGi bundle that contains a library of common code, the best practice is to create a separate OSGi bundle for the library. This library bundle can then be deployed as a middleware bundle, which is activated at the JVM server startup, or as a CICS bundle, which is installed first in the CICS system definition data set (CSD) group list processing.

4.6.4 Updating Java applications

During the development process, it is necessary to create new versions of your Java applications. OSGi package versioning and the lifecycle layer of the framework make this a flexible process. The following three key strategies can be deployed to update the Java application in CICS:

1. Refresh
   a. Deploy a new version of the OSGi bundle JAR file to the same location on zFS.
   b. Disable and discard the CICS bundle resource.
   c. Reinstall and reenable the CICS bundle resource.

This has the benefit of being the simplest and quickest method for installing new versions of the application. However, it does not provide the ability to phase in any changes without disrupting the services already deployed.
2. Version
   a. Create a version of the OSGi bundle and deploy to a new location on zFS.
   b. Disable existing bundle resources.
   c. Define and install a new CICS bundle definition that refers to the new location.

This process makes it easy to toggle between two different versions of the application by disabling and enabling CICS bundles.

3. Alias
   a. Create a version of the OSGi bundle with a new alias name for the OSGi service and deploy to a new location on the zFS.
   b. Define and deploy via a new CICS bundle definition.
   c. Update the existing program definition to refer to the new OSGi service name via the JVM class parameter.

Using an alias allows two versions of the application to be active at the same time. The application can then be switched between different versions by updating the reference to the OSGi service in the program definition.

**Note:** In addition, restarting the JVM can still be used to deploy new versions of a deployed JAR file. This can be achieved by disabling and re-enabling the JVM server, or by restarting the CICS region.
This chapter provides a detailed review of the diagnostic techniques and information that can be obtained from your Java virtual machine (JVM) server environment and applications. The chapter comprises the following sections:

1. Where to find the JVM server output files and how to interpret them.
2. How to obtain JVM server statistics.
3. A guide to tuning the JVM server.
4. Useful tools and techniques to analyze the JVM such as the IBM Health Center.
5. Interactive debugging of your CICS Java applications, by using execution diagnostic facility (EDF) and a JPDA debugger.
6. Basic failure scenarios that you might encounter during the environment setup and application development.
5.1 JVM server diagnostics

In addition to the traditional CICS administration facilities such as CEMT, system programming interface (SPI), transaction monitoring, and statistics, there are a number of unique interfaces and diagnostic logs specific to the CICS JVM environment that you can use for problem determination with Java applications. This includes JVM stdout and stderr destinations, JVM dumps, Open Services Gateway initiative (OSGi) log file, and CICS JVM trace.

5.1.1 Log files

There are various error logs used by the JVM server environment on CICS. These logs vary from zFS files to traditional data definition (DD) destinations in the CICS job. The zFS files are located, by default, in the working directory of the JVM server. The working directory is generally specified by the WORK_DIR option in the JVM profile. If not specified, it is defaulted to the UNIX System Services home directory of the CICSUSER.

**MSGUSR**

Most errors that occur in a JVM server result in a message being logged to the CICS MSGUSR DD log. For errors, warnings, or abends that might have been issued, check this log first.

**SYSPRINT**

The SYSPRINT DD log is the stdout destination normally used by C-language routines. In particular, it might contain errors from the initialization of the JVM server. If omitted, individual SYSPRINT DD logs are dynamically allocated for each JVM server instance, in the pattern SYSnnnnn.

**Stdout**

After the JVM server has initialized, output from the system.out stream is written to the stdout destination set by the STDOUT option in the JVM server profile. If the file exists, output is appended to the end of the file.

For JVM servers, the default file name is <applid>.<jvmserver>.dfhjvmout, which means that a unique file is created for each JVM server and CICS region. In addition, further unique files can be made by using the &APPLID, &JVMSERVER, &DATE, and &TIME symbols.

If you specify the USEROUTPUTCLASS option in a JVM profile, the Java class named on that option handles the System.out requests instead. CICS supplies a sample class com.ibm.cics.samples.SJMergerStream, which can be used to redirect output to stdout back the MSGUSR log.
Stderr
The stderr destination is the primary location to which Java exceptions and stack traces are written. These exceptions are typically a result of errors in Java applications or components.

The location is set using the STDERR option in the JVM server profile. If the file exists, output is appended to the end of the file.

For JVM servers, the default file name is <applid>.<jvmserver>.dfhjvmserr, which means a unique file is created for each JVM server and CICS region. In addition, further uniqueness can be achieved by using the same symbols as supported for stdout.

OSGi logs
If an error occurs in the OSGi framework, an OSGi log is created in the $WORK_DIR/<applid>/<jvmserver>/configuration/ directory of the JVM server. The file extension is .log. The OSGi framework manages the size and number of log files, and by default, this is ten log files at a maximum of 1 MB each.

5.1.2 JVM server statistics
You can get further information about the JVM server by using CEMT to inquire on the JVMSERVER resource (Example 5-1), or by using the CICS Explorer JVM Servers view.

The CEMT I JVMServer command provides information such as the initial, maximum, and current heap size and the garbage collection policy that is being used by Java. Unlike pooled JVMs, the garbage collection is handled by Java automatically depending on the policy that is specified.

Example 5-1 JVMSERVER resource statistics

I JVMS
RESULT - OVERTYPE TO MODIFY
Jvmserver(DFH$JVMS)
Enablestatus( Enabled )
Purgetype( )
Jvmprofile(DFHOSGI)
Lerunopts(DFHAXR0)
Threadcount(000)
Threadlimit( 015 )
Currentheap(9138K)
Initheap(16M)
Maxheap(256M)
Gcpolicy(-Xgcpolicy:gencon)
**JVMSERVER: Resource statistics**

You can access JVMSERVER statistics online using the `EXEC CICS EXTRACT STATISTICS JVMSERVER()` command, using the STAT transaction or through the CICS Explorer JVM Servers view. JVMSERVER statistics are mapped by the DFHSJSDS DSECT.

Example 5-2 provides a summary listing of resource statistics for our JVM server, including the number of times the JVM server has been used by Java applications and thread usage.

**Note:** Summary statistics are not available online.

*Example 5-2  Sample JVM server summary resource statistics*

```
JVMSERVER STATISTICS
    JVMSERVER Name . . . . . . . . . . : PETER
    JVMSERVER JVM profile name . . . . : DFHPETER
    JVMSERVER LE runtime options . . . : DFHAXRO
    JVMSERVER use count. . . . . . . . : 231
    JVMSERVER thread limit . . . . . . : 15
    JVMSERVER current threads. . . . . : 1
    JVMSERVER peak threads . . . . . . : 2
    JVMSERVER thread limit waits . . . : 0
    JVMSERVER thread limit wait time . : 00:00:00.0000
    JVMSERVER current thread waits . . : 0
    JVMSERVER peak thread limit waits. : 0
    JVMSERVER system thread use count. : 5
    JVMSERVER system thread waits. . . : 0
    JVMSERVER system thread wait time. : 00:00:00.0000
    JVMSERVER current sys-thread waits.: 0
    JVMSERVER peak system thread waits : 0
    JVMSERVER state. . . . . . . . . . : Enabled
    JVMSERVER JVM creation time. . . . : 05/07/2012 12:21:23.3769
    JVMSERVER current heap size . . . .: 9944160
    JVMSERVER initial heap size . . . .: 16M
    JVMSERVER maximum heap size . . . : 256M
    JVMSERVER peak heap size . . . . . : 11284320
    JVMSERVER heap occupancy . . . . . : 6873256
    JVMSERVER Garbage Collection policy: -Xgcpolicy:gencon
    JVMSERVER major GC collections. . . : 0
    JVMSERVER elapsed time in major GC.: 0
```
The following list summarizes key values in our JVM server statistics data.

### Waits

The statistics report how long Java applications have waited for a free thread in the JVM server. In Example 5-2 on page 132, the JVM server thread limit is 15, and there has only ever been a maximum of two simultaneous application threads in use (peak threads). No Java applications have been suspended waiting for JVM server threads (thread limit waits).

**Tip:** If the waits are high and many tasks are suspended with a JVMTHRD wait, you can dynamically increase the value of the THREADLIMIT attribute on the JVMSERVER resource. You can increase the value up to a maximum of 256 to make more threads available to the applications.

### Heap size and garbage collection

The statistics also report the heap size and key garbage collection information about the JVM. If the heap size after garbage collection, or the heap occupancy, is close to the maximum heap size, garbage collection might be frequent. Or, conversely, if the peak heap size is much lower than the maximum heap size, you might be able to reduce the maximum heap size to save on storage.

In our data, the maximum potential heap size is 256 MB, and the heap size after the last garbage collection (occupancy) is approximately 6.8 MB. The heap has been expanded up to 11.2 MB. Therefore, it looks like there is plenty of space allocated to the heap in our JVM server.

**Tip:** The JVM heap storage is dynamically managed by the JVM and the garbage collector. Therefore, the current heap value (9.9 MB in our data) is a transitory value that should not be used for tuning purposes. For further details on tuning the JVM heap settings, see 5.3.3, “Tuning JVM heap usage and garbage collection” on page 139.

The statistics also report major and minor garbage collection events. Minor garbage collections occur only when using the generational concurrent (gencon)
policy, and occur more frequently than major collections. When using the gencon policy, the heap is split into new and old segments. Long lived objects are promoted to the old space (tenured area), whereas short lived objects are garbage collected quickly by minor collections of the nursery area.

**System threads**
The statistics report the system thread use in the JVM server. System threads are used to collect statistics and are also used by the inquire and browse commands, but not by applications. You can determine how many times the JVM server was accessed for information, and if there were any waits for the system thread. In our data, there were five operations using the system thread *(system thread use count)*.

For more information about the statistics options, see *CICS Performance Guide*, SC34-7177-01.

### 5.2 Dumps and traces

There are several sources of detailed diagnostic information that can be collected from a JVM server. This includes several types of Java dumps, and CICS tracing for the Java Virtual Machine Domain (SJ) and application program (AP) components.

#### 5.2.1 Java dumps

JAVADUMP files are written to the JVM's working directory on z/OS UNIX, and the more detailed Java TDUMPS are written to the file named by the JAVA_DUMP_TDUMP_PATTERN option. There are three important types of Java dumps that can be produced by the JVM: Java dumps, system dumps, and heap dumps. The circumstances under which these dumps are produced are set by the JVM, and can be tailored by using the JAVA_DUMP_OPTS variable.

**Java dumps**

A Java dump (or Javacore) is a snapshot in time of the JVM, and summarizes the state of the JVM at the instant the signal occurred that caused the dump. It includes detailed information about the Java threads, along with information about memory allocation, locks, class loaders, and shared class usage. It is written in human readable form to the file javacore.%Y%m%d.%H%M%S.%pid.%seq.txt.
System dumps (cores)
A Java system dump (or core dump) is a binary dump of the process memory. System dumps are usually large and are produced by the JVM only when the JVM fails unexpectedly because of a general protection fault (GPF) or a major JVM or system error. System dumps are primarily designed for use by IBM service personnel and are in a machine-specific format, and by default are written to the file, core.%Y%m%d.%H%M%S.%pid.dmp.

Heap dumps
The JVM generates a heap dump when the heap is exhausted, or on request by the user. A heap dump contains all of the live objects on the heap when the dump is taken, and is useful for performing detailed analysis of heap usage and garbage collection. The default heap dump is not human readable, and should be processed using third party tools. By default it is written to the file, heapdump.%Y%m%d.%H%M%S.%pid.phd.

Tip: The easiest way to analyze heap usage in the JVM is by using the health center, as described in 5.4, “JVM health center” on page 140.

For more details, see Diagnostics Guide for IBM SDK and Runtime Environment Java Technology Edition, Version 6:

5.2.2 CICS tracing
In addition to the tracing produced by Java, CICS provides some standard trace points in the SJ (JVM) and AP (Application) domains for 0, 1, and 2 trace levels. These trace points trace the actions that CICS takes in setting up and managing JVM servers, installing and running OSGi applications, and making Java class library for CICS (JCICS) calls. You can activate the SJ and AP domain trace points at levels 0, 1, and 2 using the CETR transaction, or via the STNTRxx SIT parameters. The trace output from CICS internal Java code is written to a zFS file controlled by the JVMTRACE setting in the JVM server profile, whereas trace output from CICS non Java code is written to the standard generalized trace facility (GTF) or auxiliary trace data sets.

This JVM server profile setting JVMTRACE specifies the name of the z/OS UNIX file to which CICS Java tracing is written. If no value is specified, the trace is written to the default file applid.jvmserver.dfhjvmtrc which has the effect that CICS automatically creates unique output files for each JVM server using the &APPLID; and &JVMSERVER; symbols. This file is created in the directory specified by the WORK_DIR option.
**SJ and AP component tracing**

The SJ trace for JVM servers controls the tracing for the startup and shutdown of JVM servers and the initialization of the OSGi framework as follows:

- At a trace level of 0, only errors are written to the zFS trace file.
- At a trace level of 1, information, warning, and errors are written to the zFS trace file.
- At a trace level of 2, debug, information, warning, and errors are written to the zFS trace file.

In addition, the AP component controls tracing of JVM server threads, including activities of the system thread such as install, enable, disable, and discard of OSGi bundles. Output is written to the same zFS file as used for SJ tracing:

- At a trace level of 0, the AP domain does not write out information to the zFS trace file.
- At a trace level of 1, the AP domain writes out basic information about method entry and exit for Java components to the zFS file.
- At a trace level of 2, the AP domain writes out more detailed debug information about Java components and JVM server thread usage to the zFS file.

**Managing tracing for JVM servers**

When you start a JVM server, CICS appends trace entries to the JVMTRACE file if it already exists, and creates a file in zFS if it does not. To manage the size of the zFS trace file, you can delete old entries. If you disable the JVM server resource, you can delete the file or rename the file if you want to retain the information separately.

**Note:** To optimize performance, and to avoid creating large trace files in zFS, we recommend that AP tracing activity is disabled (set to 0) for production JVM servers, unless a problem is being investigated.

### 5.3 JVM tuning

If you must investigate the performance of your Java application, there is a simple checklist to follow:

1. Analyze your Java applications to ensure that they are running efficiently and do not generate too much garbage. IBM has several tools that can help you analyze the efficiency and performance of particular methods and the
application as a whole. For further details, see 5.4, “JVM health center” on page 140.

2. Tune the JVM server. You can use CICS statistics and the JVM health center to analyze the storage settings, garbage collection, task waits, and other information to investigate the performance of the JVM.

3. Optimize just-in-time (JIT). The JIT compiler is a standard part of the JVM, and continually optimizes your application code at run time and provides many benefits. However, it requires a certain amount of processor resources and elapsed time to reach the maximum optimization levels. Furthermore, any JITed code is lost when the JVM is restarted. For this reason, it is best to restart JVM servers as infrequently as possible to make the most of the JIT compiler optimizations.

4. Enable the class cache. Enabling the JVM shared class cache can improve the speed and efficiency of the JITing process. This is because the JIT can store profiling information and partially optimized code (termed AOT byte codes) in the shared cache. For further details, see 10.5.1, “Enabling the shared class cache” on page 359.

Minimum overall processor usage
This goal prioritizes the most efficient use of the available processor resource. If a workload is tuned to achieve this goal, the total use of the processor across the entire workload is minimized, but individual tasks might experience high processor consumption. Tuning for the minimum overall processor usage involves specifying large storage heap sizes for your JVMs to minimize the number of garbage collections.

Minimum application response times
This goal ensures that an application task returns to the caller as rapidly as possible. This goal might be especially relevant if there are service level agreements to be achieved. If a workload is tuned to achieve this goal, applications respond consistently and quickly, though a higher processor usage might occur for garbage collections. Tuning for minimum application response times involves keeping the heap size small and possibly using the gencon garbage collection policy.

Minimum storage usage
This goal prioritizes reducing the amount of storage used by JVMs. JVMs use 64-bit storage so it is possible to run many pooled JVMs or JVM servers in a CICS region. If pooled JVMs use a smaller storage heap, it might be possible to run more of them in the CICS region. However, choosing this goal might increase processor use. Tuning JVMs to minimize the storage heap size might result in a greater frequency of garbage collection events.
5.3.1 Examining processor usage by JVM server

You can use the CICS monitoring data to analyze the CPU time used by transactions running in a JVM server. All user tasks in a JVM server run on T8 task control blocks (TCBs), while they run Java code or issue thread-safe CICS commands. However, they might also run on L8 or the QR TCB when they run Java Database Connectivity (JDBC) or non-thread-safe CICS commands. You can use the DFHSMOLS utility to print the System Management Facilities (SMF) records or use a tool such as CICS Performance Analyzer to analyze the SMF records. In particular, the following fields from the performance data in group DFHTASK (Table 5-1) provide useful information.

Table 5-1 Processor-related monitoring fields

<table>
<thead>
<tr>
<th>Field ID</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>T8CPUT</td>
<td>The processor time during which the Java task was dispatched on a CICS T8 mode TCB.</td>
</tr>
<tr>
<td>401</td>
<td>JVMTHDWT</td>
<td>The elapsed time that the Java task waited to obtain a JVM server thread because the CICS system had reached the thread limit for a JVM server.</td>
</tr>
</tbody>
</table>

5.3.2 Tuning JVM server startup in a sysplex

If you have problems starting many JVM servers at the same time across CICS regions in a sysplex, you can improve performance by tuning the environment.

When a JVM server starts, it loads a set of libraries from the USSHOME zFS location (/usr/lpp/cicsts/cicsts42/lib). If you start many JVM servers at the same time, it can take a while for each JVM to load the required libraries. Some JVM servers might time out or take a long time to start. To reduce the startup time, you can tune the environment by using the following actions:

1. z/OS V1R13 provides significantly improved zFS I/O performance for access to sysplex shared zFS. If you are not at this level, mount the zFS data set in read-only mode to improve the time it takes to access the libraries from different JVM servers in the sysplex. Or, mount the zFS data set locally, to provide a local copy of the libraries to a CICS region. For more information, see the Redbooks publication, *Distributed File Service zSeries File System Implementation* z/OS V1R13, SG246580.

2. Create a shared class cache for the JVM servers. This enables the JVM classes to be loaded once and to be cached in memory. To use a shared class cache, add the -Xshareclasses option to the JVM profile of each JVM server. Refer to 10.5.1, “Enabling the shared class cache” on page 359, for further details.
3. Increase the timeout value for the OSGi framework. The DFHOSGI JVM profile contains the OSGI_FRAMEWORK_TIMEOUT option that specifies the amount of time that CICS waits for the JVM server to start and shut down. If the time is exceeded, the JVM server fails to initialize or shut down correctly. The default value is 60 seconds, so increase this value to a more suitable number of seconds for your environment.

5.3.3 Tuning JVM heap usage and garbage collection

Garbage collection (GC) is the process used by the JVM to free up unused objects and references from the heap storage, and to expand or compress the heap if necessary. In the JVM server environment, this process is handled by using both the T8 thread and a set of GC background threads. The GC process involves a small period when the JVM activity must be halted so that the garbage collector has exclusive access to the heap. This period is known as stop the world, and pauses any CICS tasks that are running during this time.

The aim of tuning GC is to find a compromise between reducing the duration of the stop the world interval, while at the same time minimizing the total processing time, the frequency of garbage collections, and the impact of the heap storage requirements on the JVM application server. This might seem like an impossible task, but there is a wealth of information and tools available to help in this process. Detailed information for the CICS JVM server environment is available in the IBM Redpaper™, IBM CICS Performance Series: CICS TS V4.2 and Java Performance, REDP4850. In addition, the IBM Health Center (see 5.4, “JVM health center” on page 140) provides a simple, graphical, and low impact means of analyzing the JVM.

Garbage collection policies
The JVM server in CICS TS V4.2 now uses the generational concurrent (gencon) policy by default, rather than the optthrput model used in the JVM pool environment. When using the gencon policy, the heap is split into new and old segments. Long-lived objects are promoted to the old space (tenured area), whereas short-lived objects are garbage collected more quickly by minor collections of the nursery area. The gencon policy helps to minimize the time that is spent in any garbage collection pauses and is generally considered more efficient for applications that produce many small short-lived objects.

Checklist
As a simple initial checklist for tuning, we advise collecting the following information:

1. Collect JVM server and dispatcher statistics over an appropriate interval. The JVM server statistics can tell you how many major and minor garbage
collections take place and the elapsed time that the processor spent in garbage collection. The dispatcher statistics can tell you about processor usage for T8 TCBs across the CICS region.

2. Use the dispatcher TCB mode statistics for T8 TCBs to determine how much processor time is spent on JVM server threads. The Accum CPU Time/TCB field shows the accumulated processor time taken for all the TCBs that are, or have been, attached in this TCB mode. The TCB Attaches field shows the number of T8 TCBs that have been used in the statistics interval. Use these numbers to work out approximately how much processor time each T8 TCB has used.

3. Use the JVM server statistics to find the percentage of elapsed time that is spent in garbage collection. Divide the time of the statistics interval by how much elapsed time is spent in garbage collection. In a busy system, aim for less than 2% of elapsed time used in garbage collection. If the percentage is higher than 13%, the JVM attempts to expand the heap. Increasing the maximum heap (Xmx) means that garbage collection occurs less frequently.

4. Compare the maximum heap size field with the GC heap occupancy field to determine how much live data is being used in the heap. If the heap occupancy after a garbage collection is greater than 50% of the maximum heap, consider increasing the maximum heap size, or reducing the workload in the JVM.

5. Divide the heap freed value by the number of transactions that have run in the interval to determine the average amount of garbage per transaction. You can determine how many transactions have run by looking at the dispatcher statistics for T8 TCBs. Each thread in a JVM server uses a T8 TCB.

5.4 JVM health center

The information given in this section introduces the IBM Monitoring and Diagnostic Health Center tool and provides basic guidance to help you tune JVMs in a CICS environment. The outcome of your tuning can vary depending on your Java workload, the maintenance level of CICS, the IBM SDK for z/OS, and other factors. For more detailed information about the storage and garbage collection settings, and the tuning possibilities for JVMs, download the IBM Developer Kit and Runtime Environment, Java Technology Edition, Diagnostics Guide from the following website:

5.4.1 Overview

The IBM Monitoring and Diagnostic Tools for Java - Health Center is a no-charge low-overhead diagnostic tool for monitoring an application running on an IBM Java virtual machine. It provides recommendations and analysis that help you improve the performance and efficiency of your application. Health Center can save the data obtained from monitoring an application and load it again for analysis later.

The key features and benefits of Health Center are as follows:

- **Performance analysis**
  - Java method profiling: Health Center uses a sampling method profiler to diagnose applications showing high CPU usage. It is low overhead, which means there is no need to specify which parts of the application to monitor in advance. The Health Center simply monitors everything. It works without recompilation or byte code instrumentation and shows where the application is spending its time, by giving full call stack information for all sampled methods.
  - Lock analysis: Synchronization can be a big performance bottleneck on multi-CPU systems. It is often difficult to identify a hot lock or assess the impact that locking is having on your application. Health Center records all locking activity and identifies the objects with the most contention. Health Center analyses this information, and uses it to provide guidance about whether synchronization is impacting performance.
  - Garbage collection (GC): The performance of GC affects the entire application. Tuning GC correctly can potentially deliver significant performance gains. Health Center identifies where garbage collection is causing performance problems and suggests more appropriate command-line options.

- **Memory usage analysis**
  The Health Center can identify whether your application is using too much memory, or where memory leaks occur. It then suggests solutions to memory issues, and also Java heap sizing guidance.

- **File input and output analysis**
  Health Center uses an I/O perspective to monitor application input or output (I/O) tasks as they run. You can use this perspective to monitor how many and which files are open, and to help solve problems such as when the application fails to close files.

- **Object allocation analysis**
  Health Center allows you to view the size, time, and code location of an object allocation request that meets specific threshold criteria.
The Health Center tool is provided in two parts:

1. The Health Center client is a graphical user interface (GUI)-based diagnostics tool for monitoring the status of a running JVM, installed within the IBM Support Assistant (ISA) Workbench.

2. The Health Center agent provides the mechanism by which the Health Center client obtains information about your Java application. The agent uses a small amount of processor time and memory and must be manually installed in an IBM JVM.

Figure 5-1 shows where the Health Center client and agent are located, when installed in ISA and the JVM.

![Figure 5-1 Location of Health Center client and agent](image)

### 5.4.2 Installation and configuration

In this section, we describe the installation and configuration process of Health Center.

**Install the Health Center client**

Run the following instructions to install the Health Center client:

1. Download and install the IBM Support Assistant Workbench.
   

2. Install the Health Center in the ISA Workbench.
   
   a. Follow the steps in the technote, *How to download, install, and run tools through IBM Support Assistant Workbench 4.1*, at this website:
      
When you arrive at the *Find new tools add-ons* wizard, type **health** in the search box. Click **Next** at JVM-based Tools to show matching tools. See Figure 5-2.

![Find new tools add-ons](image)

**Figure 5-2  Find for the Health Center add-on**

b. Select **IBM Monitoring and Diagnostic Tools for Java - Health Center** from the list, then click **Next**.

c. Complete the wizard by following the on-screen instructions.

d. ISA prompts you to restart the Workbench. Select **Yes** to restart.

**Start the Health Center client**

Run these procedures to start the Health Center client:

1. Start the ISA Workbench and click **Analyze Problem** from the Welcome page.

2. On the Analyze Problem tab, select **Tools** (see Figure 5-3).

![Collect Data - IBM Support Assistant Workbench](image)

**Figure 5-3  Where to find the installed tools in the ISA Workbench**
3. You are presented with a list of all the installed tools. Select **IBM Monitoring and Diagnostic Tools for Java - Health Center** and then click **Launch**. The Health Center application now starts and the Health Center: Connection wizard is displayed.

**Enable your application for monitoring**
Run the procedures in the following list to enable your application for monitoring:

1. On the first page of the Health Center: Connection wizard, click the **Enabling an application for monitoring** link, as shown in Figure 5-4.

![Health Center: Connection wizard](image)

*Figure 5-4  Health Center Connection wizard application enable link*

2. Follow the help topic displayed for your application and JDK version, and choose a port that is deallocated on z/OS; we chose port 8688.

   Add the **-Xhealthcenter:port=8688** parameter to your JVM server profile.

3. You must input the port for the Health Center agent and confirm the range of ports for searching (see Figure 5-5 on page 145).
Figure 5-5 Specify the host and port for the Health Center agent

4. Select the port that you want from the wizard and click Finish, as shown in Figure 5-6. The wizard connects to the Health Center agent.

Figure 5-6 Select a detected port for the JVM
5.4.3 Analyzing Java applications by using IBM Health Center

To analyze the performance of a Java application, you can use IBM Health Center to analyze the application, and determine areas of the application code that might be causing performance problems.

When analyzing the application, start by running the application in a JVM on its own. If you are running a mixed workload in a JVM server, it might be more difficult to analyze a particular application. Therefore, it is best to start with a single Java application running in a dedicated JVM server.

Run the procedures in the following list:

1. Add the required connection options to the JVM profile of the JVM server. The IBM Health Center documentation describes what options you must add to connect to the JVM from the tool.

2. Start IBM Health Center and connect it to your running JVM. IBM Health Center reports JVM activity in real-time, so wait a few moments for it to monitor the JVM.

3. Select the Profiling link to profile the application. You can check the time spent in different methods. Check the methods with the highest usage to look for any potential problems.

4. Select the Locking link to check for locking contentions in the application. If the Java workload is unable to use all of the available processor, locking might be the cause. Locking-in the application can reduce the amount of parallel threads that can run.

5. Select the Garbage Collection link to check the heap usage and garbage collection. The Garbage Collection tab can tell you how much heap is being used and how often the JVM pauses to perform garbage collection:

   a. Check the proportion of time spent in garbage collection. This information is presented in the Summary section. If the time spent in garbage collection is more than 2%, you might need to adjust your garbage collection.

   b. Check the pause time for garbage collection. If the pause time is more than 10 milliseconds, the garbage collection might be having an effect on application response times.

   c. Divide the amount of garbage collection by the number of transactions to determine approximately how much garbage is produced by each

   Tip: The Analysis and Recommendations tab can identify particular methods that might be good candidates for optimization.
transaction. If the amount of garbage seems high for the application, you might have to investigate the application further.

After you have analyzed the application, you can use this application to modify your application or to tune the Java environment for your workload.

5.5 Interactive debugging

In this section, we describe the technique for interactively debugging a Java program running in CICS.

5.5.1 Execution diagnostic facility

You can use the execution diagnostic facility (EDF) to test an application program online without modifying the program or the program-preparation procedure. The CICS EDF is supported by the CICS supplied transaction, CEDF, for debugging via a 3270 terminal or connection.

EDF intercepts the execution of CICS commands in the application program at various points, which allows you to see what is happening. Each command is displayed before execution, and most are displayed after execution is complete. Screens that the application program sends are preserved, so you can converse with the application program during testing, just as a user does on a production system.

_CICS Transaction Server for z/OS V4.2: Application Programming Guide_, SC34-7158-01, has more information about EDF and user instructions for EDF.

Invocation CEDF: You can also start CEDF indirectly through another CICS supplied transaction, CEDX, which enables you to specify the name of the transaction that you want to debug. When this section refers to the CEDF transaction (for example, when it describes CICS starting a new CEDF task), remember that the CEDX command might have started it.

When used with a JCICS program, CEDF intercepts the CICS commands that the Java Native Interface (JNI) programs start, which provide the interface between JCICS classes and CICS. There might not be an obvious relationship between the JCICS class method and the CICS command. For example, the HelloCICSWorld sample uses the Task.out.println method to send a message to the user terminal. The JNI program starts the SEND command to write to the panel.
Example 5-3 shows the CEDF panel for the intercepted SEND command for the following Java `println()` method:

```java
Task.getTask().out.println("Hello from a Java CICS application");
```

**Example 5-3  Writing to the panel by using the Task.out.println method**

```
TRANSACTION: JHE2 PROGRAM: DFJ$JHE2 TASK: 0000114 APPLID: EPRED2
STATUS: ABOUT TO EXECUTE COMMAND
EXEC CICS SEND
   FROM ('. DHello from a Java CICS application..')
   LENGTH (39)
   WAIT
   NOHANDLE
```

### 5.5.2 Java Platform Debugger Architecture

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.

Various third party debugging tools are available that use JPDA and can be used to attach to and debug a JVM that is running a 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, and examining the values of variables.

When you start the JVM in debug mode, the JVMDI interface is activated and more 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 breakpoints or to examine the values of variables in the application. These commands are activated by the listener and event handler threads in the JVM.

You can use any tool that supports JPDA to debug a Java application running in CICS. We use the debugger that is provided in IBM Rational Developer for System z to show how to attach a debugger to a CICS JVM. We chose the HelloCICSWORLD sample program in the getting started chapter as the application to be debugged:

1. Add the `-Xdebug` option to the JVM profile to start the JVM in debug mode. If the JVM profile is shared by more than one application, you can use a different JVM profile for debugging.
2. Add the `-Xrunjdwp` option to the JVM profile to specify the details of the connection between the debugger and the JVM. We used the following options:

```
Xrunjdwp:transport=dt_socket,server=y,address=27888,suspend=
```

The `suspend=y` tells the CICS JVM not to wait until the debugger is attached to begin execution, and the address 27888 is the port that the CICS JVM server listens on for JPDA requests.

**Note:** When you set these options in a JVM profile, any CICS Java program that runs in the JVM server runs in debug mode and waits for an attachment from, or attempts to attach to, a debugger. Instead of configuring any of the CICS supplied sample profiles for debugging, create a separate JVM server with a specific JVM profile specifically for debugging use.

Then, from the Eclipse integrated development environment (IDE) environment, perform the following actions:

3. Open the Debug perspective, and open the `HelloWorld.java` class in the `com.ibm.cics.server.examples.hello` project and set a breakpoint by clicking the left margin at a given line.

4. Within the Debug perspective, then make the following selections (see Figure 5-7 on page 150):

   a. Select **Run** → **Debug configurations** → **Remote Java Application** → **New** and create a new debug configuration, as shown in Figure 5-7 on page 150.

   b. Select **Browse** to find the project in your workspace for the `com.ibm.cics.server.examples.hello` sample containing the `HelloWorld` class.

   c. Select **Standard (Socket Attach)**.

   d. In the **Host** field, enter the host name of the z/OS system. We used `wtsc66.itso.ibm.com`.

   e. In the **Port** field enter the port specified in the JVM server profile. We used 27888.
5. Click **Debug** to start the debugging session. The debugger starts in the background.

6. Enter the JHE2 transaction ID at a CICS terminal to run the HelloWorld Java program. The terminal session hangs and the remote debugger starts in Eclipse and runs to the breakpoint.

7. From the debug perspective, you can step through the code of your class, inspect and change variables, and set new breakpoints. If the source code is available, you can also interactively debug other classes that were started during the execution of your application program.

8. End the debug session. You can let the application run to completion, at which point the connection between the debugger and the CICS JVM closes.
5.6 Basic failure scenarios

This section describes the basic failure scenarios that might happen during the
JVM server setup, the deployment of user applications, and at application run
time.

5.6.1 JVM server setup failures

If you encounter problems when setting up your JVM server, you can use both
the diagnostic tools provided by CICS and by the JVM to determine the cause of
the problem. CICS provides statistics, messages, and tracing to help you
diagnose problems related to your Java applications. The IBM Java SDK
diagnostic tools give more detailed information about the internals of the JVM,
including the garbage collection and JIT daemon threads running in a JVM.

Following is a checklist, which can help you determine the cause of your JVM
server setup problems:

1. If you are unable to start a JVM server, check that your Java SDK installation
   is correct. Use the CICS messages and any errors in the dfhjvmerr file for the
   JVM to determine what might be causing the problem:
   a. Check that the correct version of the Java SDK is installed and that CICS
      has access to it in z/OS UNIX.
   b. Check that the USSHOME system initialization parameter is set in the CICS
      region. This parameter specifies the z/FS installation location for the CICS
      UNIX System Services files.
   c. Check that the JVMPROFILEDIR system initialization parameter is set
      correctly in the CICS region. This parameter specifies the location of the
      JVM server profiles on zFS.
   d. Check that the CICS region has read and execute access to the z/OS
      UNIX directories that contain the JVM profiles.
   e. Check that the CICS region has write access to the working directory of
      the JVM. This directory is specified in the WORK_DIR option in the JVM
      profile.
   f. Check that the JAVA_HOME option in the JVM profiles points to the
      directory that contains the Java SDK.
   g. Check that SDFJAUTH is in the STEPLIB concatenation of the CICS
      startup JCL.
   h. If you are using WebSphere MQ or DB2 DLL files, check that the 64-bit
      version of these files is available to CICS.
i. If you have changed DFHAXRO or DFHJVMRO to configure the Language Environment enclave, ensure that the runtime options do not exceed 200 bytes and that the options are valid. CICS does not validate the options that you specify before passing them to Language Environment. Check the SYPRINT log for any error messages from Language Environment.

2. If your setup is correct, gather diagnostic information to determine what is happening to the application and the JVM:
   a. Add PRINT_JVM_OPTIONS=YES to the JVM profile. When you specify this option, all the options passed to the JVM at startup, including the contents of the class path, are printed to SYSPRINT. The information is produced every time a JVM is started with this option in its profile.
   b. Check the SYSPRINT and MSGUSR logs for error messages.
   c. Check the dfhjvmmout and dfhjvmerr files for information and error messages from the JVM. These files are in the directory specified by the WORK_DIR option in the JVM profile. The files might have different names if the STDOUT and STDERR options were specified in the JVM profile.

3. If the application is failing or performing poorly, debug the application by using a JPDA debugger.

4. If you are getting Java out-of-memory errors, the application might have a memory leak, or the heap size might be too small. Use CICS statistics or a tool such as IBM Health Center to monitor the JVM. If the application has a memory leak, the amount of live data that remains after garbage collection gradually increases over time until the heap is exhausted. The JVM server statistics report the size of the heap after the last garbage collection and the maximum and peak size of the heap.

These are the common errors that you might encounter in the JVM server setting up process:

1. CICS startup job errors: SDFJAUTH not in STEPLIB
   If SDFJAUTH is not in the STEPLIB concatenation, you see the following error message in the CICS job log (Example 5-4).

   Example 5-4   Error message for ENCLAVE_INIT_FAILED

   MSGUSR from CICS job log
   DFHSJ0210 04/15/2012 22:15:53 EPRED2 An attempt to start a JVM for
   the JVMSERVER resource DFH$JVMS has failed. Reason code:
   ENCLAVE_INIT_FAILED.
   DFHSJ0914 E 04/15/2012 22:15:53 EPRED2 CICSRS1 JVMSERVER DFH$JVMS is
   DISABLED because the Language Environment Enclave was not created.
The CICS Information Center provides the following information about the message DFHSJ0210 showing the cause of the problem:

The Language Environment enclave failed to initialize successfully. Check that the SDFJAUTH library is in the STEPLIB concatenation of the CICS region.

2. JVM profile errors

a. Insufficient authority to read the JVM profile

If the CICS region's user ID does not have sufficient authority to read the zFS file containing the JVM profile, the JVM server fails to start. See Example 5-5.

Example 5-5 Insufficient authority to read JVM profile

MSGUSR from CICS job log
DFHSJ0210 04/15/2012 23:47:52 EPRED2 for the JVMSERVER resource DFH$JVMS has failed. Reason code: JVMPROFILE_ERROR.

SYSPRINT from CICS job log
2012/04/16 23:47:52.577181 UTC DFH$JVMS *Exc*: Attempt to open JVM profile DFHJVMOS has failed. Runtime error message = EDC5111I Permission denied.

b. Incorrect JAVA_HOME

If you set up the JAVA_HOME with a non-existing directory, you see the error in Example 5-6.

Example 5-6 Error message for non-existing JAVA_HOME

MSGUSR from CICS job log
DFHSJ0210 04/15/2012 23:37:52 EPRED2 An attempt to start a JVM for the JVMSERVER resource CPLXJVMS has failed. Reason code: JVMPROFILE_ERROR.

DFHSJ0914 E 04/15/2012 23:37:52 EPRED2 CICSRS1 JVMSERVER CPLXJVMS is DISABLED because the Language Environment Enclave was not created.

SYSPRINT from CICS job log
2012/04/15 23:37:52.577181 GMT *Exc*: Problem encountered on line 39 of the JVM profile DFHJPDA = Failed to open JAVA_HOME. EDC5129I No such file or directory.

c. Incorrect Java version

If you set up the JAVA_HOME with an incorrect Java version, you see the error in Example 5-7 on page 154.
Example 5-7  Error message for incorrect Java version

MSGUSR from CICS job log
DFHSJ0210 04/15/2012 23:47:41 EPRED2 An attempt to start a JVM for the JVMSERVER resource CPLXJVMS has failed. Reason code: JVMPROFILE_ERROR.
DFHSJ0914 E 04/15/2012 23:47:41 EPRED2 CICSRS1 JVMSERVER CPLXJVMS is DISABLED because the Language Environment Enclave was not created.

SYSPRINT from CICS job log

d. Using CLASSPATH parameters in a JVM server

In this situation, the JVM server can be enabled but will not support OSGi. Therefore, when you install an OSGi bundle into this JVMServer, you see the error message as in Example 5-8.

Example 5-8  Error message for usage of the CLASSPATH parameter

MSGUSR from CICS job log
DFHSJ1100 04/15/2012 23:54:51 EPRED2 An attempt to install an OSGi bundle into JVM server CPLXJVMS has failed. OSGi bundle symbolic name com.ibm.cics.server.examples.hello, version 1.0.0, reason code JVMSERVER_NOT_OSGI_ENABLED.
DFHRL0102 E 04/15/2012 23:54:51 EPRED2 CEDA The CICS resource life-cycle manager failed to create the resource hello and returned with reason CALL_BACK_ERROR.

To address this issue, do not use CLASSPATH-related parameters in your JVM profile when using the OSGi JVM server environment.

5.6.2 Deploying from Explorer SDK failures

You might have problems when deploying your Java application using CICS Explorer SDK. In this section, we describe some of the common errors when deploying from Explorer SDK:

- Unable to deploy CICS bundle project

  If there are errors in the CICS bundle project in your Eclipse IDE, the CICS Explorer SDK will not export files to UNIX System Services. Following are some of the typical errors:
a. Check that your main class does actually have a `public static void main(String[])` method or uses the `public static void main(CommAreaHolder)`

b. Check that the OSGi project symbolic name and version in the .osgibundle file in the CICS bundle project matches the details in the OSGi plug-in project manifest.mf

- **Include package error**

In the development process, you might need to use Java ARchive (JAR) files provided by third parties (such as the WebSphere MQ or DB2 drivers). Ensure that the JAR files that you require are OSGi-enabled, and deployed either as middleware bundles or CICS bundles. If the dependant bundles are not available, you see the following error on bundle activation, because the OSGi framework will not be able to import the dependent packages, as in Example 5-9.

*Example 5-9  Include package error*

**Configuration:**
Import-Package: com.ibm.cics.server;version="[1.300.0,2.0.0)",
com.ibm.db2.jcc

**Error:**
No available bundle exports package 'com.ibm.db2.jcc'

This problem should be solved by ensuring the dependant bundles are available before the CICS bundle is enabled.

- **Incorrect value for the Bundledir attribute in bundle resource**

You might get the error message that is shown in Example 5-10.

*Example 5-10  Wrong Bundledir specified in bundle*

**Error shown in Explorer**
CNX0591E: EXEC CICS command(CSD INSTALL) RESP(INVREQ) RESP2(633)

**Error shown in CEDA**
S The installation of BUNDLE EXMPBUN1 failed because a manifest was not found in the bundle root directory.

**MSGUSR from CICS job log**
DFHRL0103 E 04/02/2012 08:46:02 EPREDB CWWU The CICS resource life-cycle manager failed to create the BUNDLE resource EXMPBUN1 because the manifest /u/cicsrs5/EPREDB/JavaApps/META-INF/cics.xml specified in the bundle root directory was not found.
To fix this problem, you must set the bundle resource Bundledir attribute to the correct zFS directory where the CICS bundle project was deployed.

- Middleware bundle prevents JVM server activation

A non-OSGi enabled JAR file was installed as a middleware bundle. The JVM server fails to enable the OSGi framework and in the DFHJVMTRC file, there is an error running the setup class for the middleware bundle. See Example 5-11.

**Example 5-11  Active bundles with null name**

MSGUSR from CICS job log
DFHSJ0215 04/05/2012 22:49:39 EPRED3 The JVM Server JDBCJVMS failed to initialize the OSGi framework. The JVM will be terminated.

**Note:** To avoid this failure, you must ensure the JAR files that you are using in the OSGI_BUNDLES list are OSGi enabled.

- JVM server installation times out

For OSGi only, use the OSGI_FRAMEWORK_TIMEOUT option to specify how many seconds CICS waits for the OSGi framework to initialize or shut down before timing out. The default value is 60 seconds. If the framework takes longer than the specified time, the JVM server fails to initialize or shut down correctly. For further advice on resolving startup issues with the JVM server, see 5.3.2, “Tuning JVM server startup in a sysplex” on page 138.

- Installation resource error

In the CICS application deployment process, it is possible to put all related resources, such as the program, transaction, JVM server, and bundle in the same CICS system definition data set (CSD) group. If you install this whole group, you get some installation failure messages during the first phase of the bundle installation. However, when you display the resources, all the resources are installed correctly. See Example 5-12.

**Example 5-12  Install group error message**

CEDA install message
W BUNDLE MYJELLO has been installed as disabled because one or more of its associated resources failed to install.

MSGUSR from CICS job log
This is not a functional problem, in the sense that the JVM server and OSGi bundle install successfully. However, CEDA issues this warning message because at the time the bundle is installed, the JVM server is not yet active.

5.6.3 Java application runtime failures

In this section, we describe some of the common errors you might encounter when you try to run your Java application:

- Abend AJ04

  If you receive abend AJ04, this indicates CICS cannot find your program’s main method or an unhandled exception has been thrown in the Java environment. This is a common error for Java applications. See Example 5-13.

  **Example 5-13  Most common abend AJ04**

  DFHAC2206 08:02:55 EPRED2 Transaction HELO failed with abend AJ04. Updates to local recoverable resources backed out.

  The abend AJ04 is typically caused by one of the following problems:

  - The name of the main class or OSGi service alias on the program definition is incorrect.
  - The bundle that references the OSGi bundle is not enabled.
  - The OSGi bundle manifest does not have the correct CICS-MainClass attribute that refers to the fully qualified package and class name, or alias, of the main-method Java class. For further details, see 6.2, “Sample Java application code” on page 170.
  - The bundle has been installed via the OSGI_BUNDLES parameter rather than via a CICS bundle definition. OSGI_BUNDLES is for middleware bundles, not application bundles. Therefore, CICS-MainClass entries are not processed by CICS for OSGi bundles installed this way.
Incorrect output or behavior

If you determine that your program is not behaving as expected, check the log files to see if any exceptions are thrown by your application. You might also debug your program interactively to determine its operation. See 5.5, “Interactive debugging” on page 147” for details.

No response

In this case, your program might be waiting on a JVM server thread, waiting on a CICS resource such as a file or queue, which is looping, or deadlocked within the application.

- If it is waiting on a CICS resource via a JCICS command, standard CICS monitoring and SPI facilities can be used to inquire on the task wait.
- If it is waiting on a JVM server thread to become available, this will also be shown in the SPI as a wait of type JVMTHRD.
- If it is looping through CICS code that is running a JCICS command, CICS monitoring and SPI commands can be used to view the accumulation in CPU time. It will also be subject to runaway task purging if set on the RUNAWAY attribute on the transaction definition.
- If it is looping within Java code, CICS SPI and monitoring cannot be used to analyze the problem. And, CICS task monitoring will not show the accumulated CPU usage until the task terminates or enters a JCICS command. However, if it is accumulating CPU, this shows at the CICS region level, and the task will also be subject to runaway task purging.
- If the application is deadlocked within the Java code, a Java core dump can be used to analyze the contending threads. For further details, see 5.2.1, “Java dumps” on page 134.

java.lang.OutOfMemoryError

- If the application reports an out of memory exception, this indicates that the Java heap has been exhausted. Refer to 5.4, “JVM health center” on page 140, about how to assess and respond to memory usage problems.
Application development

In this part of the publication, application development is described. We introduce the CICS Explorer software development kit (SDK) and how it can be installed into an Eclipse integrated development environment (IDE), and then used to develop Java applications in CICS. An introduction to Java programming in a CICS environment is provided. And we describe the difference between traditional CICS programming and using CICS services from Java.
Developing a simple Java application

In this chapter, we provide a step-by-step guide on using the CICS Explorer software development kit (SDK) within Eclipse to develop a simple Java application as an Open Services Gateway initiative (OSGi) plug-in project. We then demonstrate the deployment process for this application into a CICS Java virtual machine (JVM) server.
6.1 Setting up the Eclipse IDE

The first step is to install a suitable Java Integrated Development Environment (IDE). Eclipse is an open development platform, produced by the Eclipse Foundation (www.eclipse.org), composed of extensible frameworks, tools, and run times for building, deploying, and managing software across the lifecycle. We used Eclipse 3.6, although you can use any supported environment such as Rational Application Developer V8 for WebSphere.

For full details about how to set up the Eclipse IDE and install the Explorer SDK, see Appendix A, “Installing the CICS Explorer software development kit” on page 367.

6.1.1 Set the target platform

Before embarking on any application development, we set up Eclipse to use the versions of the OSGi implementation and the CICS packages that we want to use. In our case, it is CICS Transaction Server (TS) V4.2, which uses OSGi 4.2 and the Java 6 run time. The Explorer SDK provides a simple way to set this information using a Target Platform.

To set the Target Platform, select Window → Preferences to open the Eclipse Preferences window. On the left menu, select Plug-in Development and then Target Platform. The target platform options are displayed on the right of the window, as seen in Figure 6-1 on page 163.
Figure 6-1   Eclipse Preferences window

Click **Add** to open a New Target Definition window, as shown in Figure 6-2.
Next, select **Template** and open the drop-down list labeled **Base RCP (Binary Only)**. A list is revealed which should contain the versions of CICS that the CICS Explorer SDK supports. See Figure 6-3.

![Base RCP (Binary Only) drop-down list](image)

For this example, we create a CICS TS V4.2 application. Do the following steps:

- Select **CICS TS V4.2 Runtime** from the drop-down menu
- Click **Next**
- When complete, click **Finish**

The Preferences window shows that the CICS TS V4.2 target platform has been added, as in Figure 6-4.

![Preferences window](image)

---

164 CICS and the JVM server: Developing and Deploying Java applications
Complete the process by clicking the check box for **CICS TS V4.2 Runtime**. Click **Apply** followed by **OK** to close the window.

### 6.1.2 Create a plug-in project

Now that we set the Eclipse environment, the next step is to create a project in Eclipse that will contain the code for the application. To do this, we must create a plug-in project. A plug-in project is also referred to as an **OSGi bundle project**.

- From the Eclipse menu, select **File → New → Project**.
- You are presented with the New Project window.
- Open the **Plug-in Development** submenu and select **Plug-in Project** (see Figure 6-5).

![Figure 6-5  New Project window](image-url)
Clicking **Next** displays the **New Plug-in Project** window, as shown in Figure 6-6.

- Give the project a name. We used *ITSO Plugin Project*.
- Select an **OSGI framework** and select **standard** from the drop-down menu.

![Figure 6-6 New Plug-in Project window](image)

**Note**: We could have selected the Equinox OSGi framework. However, for future portability, it is more prudent to target a generic specification than any one implementation of OSGi.

Clicking **Next** displays the final options of the New Plug-in Project dialog. See Figure 6-7 on page 167.
The following changes should then be made:

- Remove `.qualifier` from the end of the Version field. CICS does not support the use of a version qualifier. The value should read `1.0.0`.

- Select the **Execution Environment** that matches the minimum Java application programming interface (API) level required to run the application. We selected `JavaSE-1.6` based on the assumption that our application uses some Java 6 specific API. CICS TS V4.2 uses a Java 6 run time so we are confident that the dependency can be satisfied.

**Tip:** The compiler tries to compile your OSGi bundle at the level of Java runtime environment (JRE) specified in the Execution Environment header. Byte-code compiled at the JavaSE-1.6 level can run in any environment capable of running Java 6, Java 7, or later.

- Clear the **Generate an activator** check-box.

- Click **Finish**. Allow Eclipse to switch to the plug-in perspective when prompted. See Figure 6-7.
6.1.3 Edit the plug-in manifest file

The next step is to edit the plug-in manifest file and add the Java class library for CICS (JCICS) API dependencies. If you do not perform these steps and you code JCICS API, you will not be able to compile your Java application.

Complete the following steps to edit the manifest file:

1. Select the Dependencies tab, which displays a window that should show no dependencies. See Figure 6-8.

![Dependencies tab](image)
2. Click Add for Imported Packages. This selection opens the Package Selection window, as shown in Figure 6-9.

![Package Selection window](image)

Figure 6-9 Package Selection window

3. Type into the Package Selection: field com.ibm.cics.server. The box should display all matching available plug-ins. When you see the package that you require, select it by double-clicking it. In our case, we added com.ibm.cics.server (1.300.0), which is the version shipped with CICS TS 4.2 run time.

4. Add other packages as necessary. In our example, we do not need to add anything else.

5. Type Ctrl+s to save the dependency information in the manifest file.

**Note:** Adding a specific CICS import-package is somewhat restrictive because it ties an application bundle to a specific version of the CICS TS API. To make our application forwards-compatible with new CICS releases, we modified the Import-Package statement to specify a minimum of v1.300.0 and maximum of anything less than v2 using the syntax \([1.300.0,2.0.0)\). See Example 6-1.

This ensures that our application can be deployed into newer versions of the CICS OSGi run time until the API version is increased. For more information about the versioning scheme for the com.ibm.cics.server package, see Table 2-2 on page 23.

**Example 6-1 Required dependencies**

| Manifest-Version: 1.0 |
| Bundle-ManifestVersion: 2 |
| Bundle-Name: ITSO_Plugin_Project |
6.2 Sample Java application code

We have now created a plug-in project, or OSGi bundle, in which we can begin to develop Java code.

Switch to Java perspective
Switch to the Java perspective by selecting Window → Open Perspective → Other. This selection opens the Open Perspective window from where you should select Java (default), as shown in Figure 6-10.
Create the Java code
The Java code is contained in a src folder, which can be seen in the Package Explorer frame on the left of the Eclipse window. See Figure 6-11.

1. Right-click the src folder and create a package:
   - Right-click src and select **New → Package**
   - Give the package a name, for example, `com.ibm.itso.myPackage`
   - Click **Finish**

2. Within the new package, create a class:
   - Right-click the package that you just created and select **New → Class**
   - Give the class a name, for example, `MyClass`
   - Check the box labeled **public static void main(String[] args)**
   - Click **Finish**
   - The generated Java program looks like the code shown in Example 6-2. The program is currently a stub, which has no code inside the `main` class

```
Example 6-2   Generated Java sub

package com.ibm.itso.myPackage;

public class MyClass {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
}
```

Figure 6-11   Package Explorer frame
- Add import statements that are shown in Example 6-3 into the generated Java code after the package statement, but before the public class statement.

**Example 6-3  Required import statements**

```java
import com.ibm.cics.server.CommAreaHolder;
import com.ibm.cics.server.InvalidRequestException;
import com.ibm.cics.server.Task;
```

**Note:** The import statements that are shown are required to use the JCICS classes in this program. If further JCICS function is added, you need to add further import statements to specify the classes you use.

Alternatively, all of the JCICS classes can be imported with a single import statement, which reads: `import com.ibm.cics.server.*;`

3. Change `main(String[] args)` to `main(CommAreaHolder CAH)`

4. Add some code to the `main` method (which is empty). For this example, we add the code, which can be found in the sample Java CICS program HelloCICSWorld. Replace the generated comment line `// TODO Auto-generated method stub`, with the code in Example 6-4.

**Example 6-4  HelloCICSWorld Java code**

```java
String ProgName = null;
Task t = Task.getTask();
if ( t == null )
   System.err.println("HelloCICSWorld example: Can't get Task");
else
{
   try {
      ProgName = t.getProgramName();
   } catch (InvalidRequestException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
   }
   t.out.println("Hello from a Java CICS application - Progname = " + ProgName);
}
```
5. Next, we edit the manifest file to include the name of the main method that is called when the program is started by CICS. CICS supports a custom Manifest header. It is used to indicate to CICS, which main methods should be exposed outside of the OSGi framework. These main methods are automatically registered by CICS as a special type of OSGi services that can be targeted by a CICS Java Program link:

- From the Package Explorer, expand META-INF. Then, open the MANIFEST.MF file by double-clicking it or by right-clicking and selecting Open
- Click the MANIFEST.MF tab
- Add the line CICS-MainClass: com.ibm.itso.myPackage.MyClass at the bottom, see Example 6-5
- Type Ctrl+s to save the changes made

Example 6-5  Updated Manifest file

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: ITSO_Plugin_Project
Bundle-SymbolicName: ITSO_Plugin_Project
Bundle-Version: 1.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: com.ibm.cics.server;version="[1.300.0,2.0.0)"
CICS-MainClass: com.ibm.itso.myPackage.MyClass
```

Note: If there is more than one main method in the project that you want to link to from a CICS Java Program, add subsequent comma separated main class values. For example, the following declares two main classes: MyClass and MyClass2

```
CICS-MainClass: com.ibm.itso.myPackage.MyClass,
com.ibm.itso.myPackage.MyClass2
```

- Press Enter at the end of the final line, because the manifest file must end with a blank line
- Type Ctrl+s to save the changes made

6.3 Creating a CICS bundle project

Now that we have a simple Java application in the form of an OSGi bundle, we must create a CICS bundle project to contain and deploy it. Although conceptually, the CICS bundle project contains any OSGi bundles, in practice,
the CICS bundle project references those OSGi bundles because they are all peer projects in your Eclipse client. After a CICS bundle project has been deployed to zFS (also causing the referenced OSGi bundles to be deployed), a CICS bundle resource definition can be used to install its contents into the JVM server run time.

In our example we have created a single OSGi bundle which we will now add into our CICS bundle project.

### 6.3.1 Create the CICS bundle

Take the following steps to create a CICS bundle:

1. From the Package Explorer, right-click into an area of white space and select **New** → **Other** → **CICS Resources** → **CICS Bundle Project** → **Next**.
2. Give the CICS bundle project a name. See Figure 6-12.
3. Click **Finish**.

![Figure 6-12  Creating a CICS bundle project]

### 6.3.2 Add the OSGi bundle to the CICS bundle

Now that we have created the CICS bundle, we must connect to it the OSGi bundle (or bundles) which contain the Java code that we have written:

1. Right-click the CICS bundle project name created in 6.3.1, “Create the CICS bundle” on page 174.
2. Select **New → other → CICS Resources → Include OSGi Project in Bundle → Next**. This action opens the window that is shown in Figure 6-13.

![Figure 6-13 Include the OSGi bundle](image)

3. From this dialog, select the OSGi bundle that you want to include in the CICS bundle. In our case, there is only one to choose, so select it.

4. You must also enter the name of the target JVM server.

5. Click **Finish** to complete the creation of the CICS bundle project. You should now see the completed CICS bundle project with a reference to the OSGi bundle project. See Figure 6-14.

![Figure 6-14 CICS bundle project](image)
6.4 Deploying the CICS bundle

Now that we have created our CICS bundle project and referenced the OSGi bundles that contain our Java code, we deploy the CICS bundle to our z/OS system so that we can use it to create and install the required CICS resources.

6.4.1 Connect to z/OS

The first step of this process is to connect to our z/OS system using the FTP protocol so that we can upload the bundle files to the z/OS UNIX file system. We use these files later when creating and installing our CICS resources:

1. If you have already created a connection to your z/OS system:
   a. Switch to the z/OS perspective.
   b. From the Host Connections tab, connect to your z/OS system.
   c. Skip to step 4.

2. If you have not connected to your z/OS system, you must create a connection:
   a. In Eclipse, switch to the z/OS perspective.
   b. From the Host Connections tab (Figure 6-15), click z/OS FTP.
   c. Click Add. The Add z/OS FTP Connection window opens. See Figure 6-16 on page 177.
d. Enter the host name or IP address of your z/OS system into the **Host name** field.
e. Change the port number only if you are sure that the default of 21 is not being used.
f. Click **Save and Connect**, which opens the Signon window that is shown in Figure 6-17 on page 178.
6.4.2 Export the bundle to z/OS

We have successfully connected to our z/OS system. We now export the CICS bundle that we created in 6.3, “Creating a CICS bundle project” on page 173 to the UNIX System Services file system:

1. Switch back to the plug-in development perspective and right-click your CICS bundle project.
2. Select **Export Bundle Project to z/OS UNIX File System**. This opens the Export to z/OS File System dialog. See Figure 6-18.

![Figure 6-18 Export bundle to z/OS](image)

3. Edit the parent directory field to point to the UNIX System Services directory into which you will upload the OSGi bundle file. The text box for the parent directory field (Figure 6-19 on page 180) shows which directories currently exist under the parent directory.

4. Select **Clear existing contents of Bundle directory** so that the directory is cleared out before deployment.
5. Click **Finish**. The chosen directory should now exist containing some .jar files and .osgibundle files and a cics.xml file in directory META-INF. See Figure 6-19.

```
CICSRS9 @ SC66:/u/cics/JavaApplications/ITSOCICSBundle>ls -l
total 48
-rw-r----- 1 CICSRS9 SYS1 141 Sep 20 13:35 ITSO_Plugin_Project.osgibundle
-rw-r----- 1 CICSRS9 SYS1 1751 Sep 20 13:35 ITSO_Plugin_Project_1.0.0.jar
drwxr-x--- 2 CICSRS9 SYS1 8192 Sep 20 13:35 META-INF
CICSRS9 @ SC66:/u/cics/JavaApplications/ITSOCICSBundle>
```

**Figure 6-19  Bundle directory listing**

## 6.5 Installing the CICS resources

Now that we have uploaded the osgibundle and Java ARchive (JAR) files to UNIX System Services, we define the CICS resources necessary to run our Java program.

Before we can run our program, we connect to our CICS region to define and install a bundle resource, a program resource, and a transaction.

### 6.5.1 Connect to CICS

Take the following steps to connect to CICS:

1. Switch to the CICS SM perspective in Eclipse by selecting the **CICS SM** perspective icon on the Eclipse desktop. See Figure 6-20.

```
Figure 6-20  CICS SM perspective
```

2. Connect to CICS using a CICS Management Interface connection for a single region. Or, connect to the WUI region if you have a CICSPlex. If you have not connected to CICS before, the following instructions show how to create a CICS Management Interface connection:

   a. At the lower right corner of the Eclipse window, click the down arrow.
b. Select **New CICS Management Interface Connection...**, as shown in Figure 6-21.

![Manage Connections]

- New CICS Management Interface Connection ...
  - iyclzcfg:01490
  - winmvs20:01490
  - New CICSPlex SM Data Interface Connection ...

*Figure 6-21   New CICS Management Interface*

c. This opens the Add CICS Management Interface Connection window that is shown in Figure 6-22.

![Add CICS Management Interface Connection]

*Figure 6-22   Add CICS Management Interface Connection window*

d. Give the connection a name.
e. Enter the host name or IP address of the z/OS system where CICS is running.
f. Enter the port number that the CICS region is using to listen for CMCI requests. This is the port number that has been specified on the TCPIPSERVICE installed in CICS for CMCI requests.
g. Click **Save and Connect**.
6.5.2 Create and install a CICS bundle resource

In this section, a CICS bundle resource is created and installed.

Create the bundle resource

From the CICS SM perspective of CICS Explorer, perform the following instructions:

1. Select **Definitions → Bundle Definitions**. See Figure 6-23.

![Figure 6-23 Select bundle definition from the menu](image1)

This opens the Bundle Definitions window in Eclipse. See Figure 6-24.

![Figure 6-24 Bundle Definition window](image2)
2. From this window, right-click into the window and select **New**. The New Bundle Definition window that is shown in Figure 6-25 is opened.

![New Bundle Definition window](image)

**Figure 6-25  New Bundle Definition window**

3. On the New Bundle Definition window, the following fields are completed:
   - **Resource Group**: This is the group which contains the new bundle definition.
   - **Name**: The name of the bundle resource.
   - **Bundle directory**: This is the directory containing the OSGi bundle files that were uploaded in “Deploying the CICS bundle” on page 176.
   - **The Open Editor check box opens the editor, which allows you to see (and edit) the resource just created. It is not required to do this step unless you want to confirm or change the definition.

4. Click **Finish** and the bundle definition is saved.

**Install the bundle definition**

Now that the bundle resource has been created and stored, we must install the definition into our CICS region. This can be done from CICS Explorer by right-clicking the resource in the definitions window and selecting **Install**.
Figure 6-26 shows how to initiate the installation of the bundle.

<table>
<thead>
<tr>
<th>Name</th>
<th>Version</th>
<th>Description</th>
<th>Change Time</th>
<th>Change User ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>bunl</td>
<td></td>
<td>CICS bundle containing OSGI service</td>
<td>23-Sep-2012 12:30:22</td>
<td>CICSUSER</td>
</tr>
<tr>
<td>DFH305GB</td>
<td>New...</td>
<td>CICS XMLTRANSFORM Bundle for new...</td>
<td>24-Jun-2011 14:55:45</td>
<td>BRENDAH</td>
</tr>
<tr>
<td>DFH305Q8</td>
<td>New from...</td>
<td>CICS XMLTRANSFORM Bundle for new...</td>
<td>24-Jun-2011 14:55:47</td>
<td>BRENDAH</td>
</tr>
<tr>
<td>DFH305QT</td>
<td>Open</td>
<td>Catalog Manager sample eventbundle</td>
<td>24-Jun-2011 14:55:48</td>
<td>BRENDAH</td>
</tr>
<tr>
<td>EPBUN01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6-26 Installing a bundle definition

Clicking **Install** opens the Perform Operation window that is shown in Figure 6-27.

![Perform Operation window](image)

Figure 6-27 Perform Operation window

Select the CICS region into which you will install this definition and click **OK**.

The installation might create some **OSGI service** resources in the CICS region. The number of OSGI service resources created will correspond to the number of CICS-MainClass definitions in the manifest file. See 6.2, “Sample Java application code” on page 170.
Chapter 6. Developing a simple Java application

The OSGI services can be seen by opening the OSGI Services window in Eclipse. Select Operations → Java → OSGI Services from the Eclipse menu. The OSGI Services window looks something like Figure 6-28.

![OSGi Services view](image)

Make a note of the fully qualified OSGi service name because it will be required when you create your program definition. In the example in Figure 6-28, the service name is `com.ibm.itso.myPackage.MyClass`.

If your OSGi service does not appear in this window, it is likely that you did not edit the manifest file correctly in 6.2, “Sample Java application code” on page 170.

### 6.5.3 Create a CICS transaction and program resource

Everything that is needed to run our OSGi Java program has been created, except for the CICS transaction and program resources.

**Program definition**

The program definition can be created from the Eclipse environment or by using CEDA. Choose a suitable name for the program.

The important part of this program definition is the JVM attributes section, as shown in Example 6-6.

**Example 6-6  JVM Attributes**

<table>
<thead>
<tr>
<th>JVM ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>JVM         ==&gt; Yes</td>
</tr>
<tr>
<td>JVMClass     ==&gt; com.ibm.itso.myPackage.MyClass</td>
</tr>
<tr>
<td>(Mixed Case)   ==&gt;</td>
</tr>
<tr>
<td>==&gt;</td>
</tr>
<tr>
<td>==&gt;</td>
</tr>
<tr>
<td>JVMServer    ==&gt; TONYJVM</td>
</tr>
<tr>
<td>JVPProfile   ==&gt;</td>
</tr>
</tbody>
</table>
The JVM server is the name of the JVM server that runs the program and must match the JVM server named in 6.3.2, “Add the OSGi bundle to the CICS bundle” on page 174.

The JVM class must match the OSGi service name when you installed the bundle definition in “Install the bundle definition” on page 183. This name is also the same as the class name declared in the CICS-MainClass part of the manifest file.

JVM is YES

Install the program definition and then create a transaction to start this program (this action is left to the reader to choose a suitable transaction name).

6.5.4 Run the program

Run the program from a CICS terminal by typing the transaction name and by pressing Enter. The transaction should produce output as shown in Figure 6-29, confirming the Java program has worked as expected.

Figure 6-29   Successful execution
The Java CICS application programming interface

In this chapter, we give you an introduction to Java programming in a CICS environment.

First, we describe the difference between traditional CICS programming and using CICS services from Java. Then, we briefly introduce the main functional areas of the application programming interface (API) with an in-depth discussion of each. Also, we show and explain some example code to help you get started.
7.1 Introduction to Java class library for CICS

There is a fundamental difference between using the CICS API from within traditional languages, such as COBOL, PL/I, and C, and using it from Java applications. In languages other than Java, CICS commands are implemented as a language extension, that is, they are embedded into the source program bracketed by EXEC CICS...END-EXEC clauses. Therefore, the source program cannot be compiled as it is. Instead, you must first run it through the CICS translator, which extracts all CICS calls and replaces them by calls to the CICS language interface module.

This procedure produces a program that can be compiled by the respective language compiler. If you are familiar with database access using embedded SQL, you can see it is much the same concept, (Figure 7-1).

Figure 7-1 Preparing CICS programs in COBOL
In Java, however, things are much simpler. CICS commands are not
implemented as a language extension but rather as a regular Java API (the Java
class library for CICS (JCICS) API). Therefore, no additional translation steps
are necessary. In that sense, Java programming is no more difficult than using
any other CICS API.

7.2 A short overview of the JCICS API

In this section, we provide a high-level overview of the CICS API, briefly
describing its various categories. Later in this chapter, we describe most of the
API in more detail. Figure 7-2 on page 190 shows a simplified class diagram of
the JCICS API.
Figure 7-2  Overview of the JCICS API
7.2.1 Functional areas

In the following sections, we describe several functional areas of the API.

Program control
Program control services allow one program to call another program in the same unit of work. Data can be passed to and received back from the called program. Until recently this was achieved by means of a shared area of storage known as the COMMAREA, but since CICS TS V3.1, an additional mechanism known as channels and containers is available as an alternative to a COMMAREA.

File control
Using file control services you can access VSAM files, which come in three different flavors:
- Key-sequenced data set (KSDS)
- Entry-sequenced data set (ESDS)
- Relative record data set (RRDS)

Synchronization
Synchronization services provide a mechanism to ensure that shared resources cannot be modified (or even accessed) by more than one transaction at a time.

Scheduling services
Scheduling services allow a program to start other transactions asynchronously, either immediately or at some specified later time.

Unit of work
This is one of the easiest and probably most important JCICS services to use. You use these services to commit or roll back the current unit of work. The result is that updates to the current transaction are either permanent, or backed out.

There is no API to explicitly start a new unit of work. A new unit of work is considered active when the previous one ended.

To commit the current unit of work, use the Task.commit() method:
Task.getTask().commit();

To back out the current unit of work, use the Task.rollback() method:
Task.getTask().rollback();
Document services
Document services are most often used with CICS Web support services ("CICS Web support and TCP/IP services") to dynamically build HTML pages.

You can work with document templates, which are portions of a document that are created offline (or by another program). These document templates might contain symbols that represent dynamic content. You can replace those symbols to produce the actual document that is returned.

CICS Web support and TCP/IP services
CICS Web support allows CICS programs to produce dynamic web content, usually in response to a client request from a web browser. In other words, they can behave like the Common Gateway Interface (CGI) programs, or Java servlets, that are supported by most web servers.

The JCICS API provides the following web-related and TCP/IP-related services:
- Examining an HTTP request
- Sending a response back to the client
- Getting the client's host name
- Security-related services, such as authorization and certificate validation

Temporary storage queues
Temporary storage is the primary CICS facility for storing data that must be available to multiple transactions. Data items in temporary storage are kept in queues whose names are assigned dynamically by the program that stores the data. Think of a temporary storage queue that contains multiple items as a small data set whose records are addressed either sequentially or directly, by item number. If a queue contains only a single item, think of it as a named scratch-pad area.

Transient data queues
Transient data queues (TD queues) are similar in some respects to temporary storage queues, but there are some major differences:
- They cannot be created immediately; they must be predefined.
- Items can be read only sequentially and cannot be changed.
- Reading from TD queues is destructive, that is, each item can be read only once. After a transaction reads an item, that item is removed from the queue and is not available to any other transaction.

Today, TD queues are mostly obsolete because the preferred alternative is to use WebSphere MQ message queues. Therefore, we do not describe TD queues any further in this book. They are however, fully supported by the JCICS API.
Terminal control
Terminal control services allow an application program to interact with the user’s terminal by displaying data on and retrieving user input from the terminal.

Terminal control in Java is not widely used because there is no JCICS support for basic mapping support (BMS). BMS is a set of CICS services and tools to create, display, and interact with panel forms from an application program. BMS panels are much easier to use (and more portable) than direct interaction with the raw terminal.

During this residency, we developed a small package called JBMS (for Java Basic Mapping Support), which allows you to create and interact with panel forms, just like the real BMS does. Read all about it in 7.13, “Terminal services” on page 239.

7.2.2 Miscellaneous services
In this section, we cover several services that do not fall into one of the categories mentioned previously.

Specifically, we now cover the following services:

- Inquire about the system name and IBM VTAM® application ID of the CICS region by using Region.getSysId() for the system name, and Region.getAPPLID() for the VTAM application ID.
- Enable and disable tracing in the CICS region: Region.enableTrace(), Region.disableTrace().
- Examine and modify the contents of the common work area (CWA), using Region.getCWA() and Region.setCWA().
- Retrieve the transaction name and program name under which the current program is running using task.getProgramName(), which is useful for logging purposes.
- Get the name of the user who started the transaction.
- Examine and modify the transaction work area (TWA). The TWA is a small (up to 32 K) area of storage that is allocated when a transaction is initiated and is initialized to binary zeros. It lasts for the entire duration of the transaction and is accessible to all local programs in the transaction. Using the TWA is not recommended for new applications.
- Determine how the current program was started by using Task.getTask().getSTARTCODE().
Table 7-1 lists the different possible strategies and their meanings.

Table 7-1  Startcodes as returned by Task.getTask().getSTARTCODE()  

<table>
<thead>
<tr>
<th>Startcode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>The task was initiated to process a distributed programming link (DPL) command that did not specify the SYNCONRETURN option. (The task is not allowed to issue syncpoints.)</td>
</tr>
<tr>
<td>DS</td>
<td>The task was initiated to process a distributed programming link (DPL) command containing the SYNCONRETURN option. (The task is allowed to issue syncpoints.)</td>
</tr>
<tr>
<td>QD</td>
<td>CICS initiated the task to process a transient data queue that reached trigger level.</td>
</tr>
<tr>
<td>S</td>
<td>Another task initiated this one using a START command that did not pass data in the FROM option.</td>
</tr>
<tr>
<td>SD</td>
<td>Another task initiated this one using a START command that passed data in the FROM option.</td>
</tr>
<tr>
<td>SZ</td>
<td>The task was initiated with a front-end programming interface (FEPI) START command (see CICS Front End Programming Interface User’s Guide, SC34-6234, for further information).</td>
</tr>
<tr>
<td>TO</td>
<td>The task was initiated to process unsolicited input from a terminal (or another system), and the transaction to be executed was determined from the input.</td>
</tr>
<tr>
<td>TP</td>
<td>The task was initiated to process unsolicited input or in response to a RETURN IMMEDIATE command in another task. In either case, the transaction to be executed was preset (in the RETURN command or in the associated TERMINAL definition) without reference to input.</td>
</tr>
<tr>
<td>U</td>
<td>CICS created the task internally.</td>
</tr>
</tbody>
</table>

7.2.3 Services that the JCICS API does not support

JCICS does not support these CICS services:

- Advanced Program-to-Program Communication (APPC) unmapped conversations
- CICS Business Transaction Services
- DUMP services
- Journal services
- Storage services
- Timer services
- Basic mapping support (BMS) services: SEND MAP and RECEIVE MAP
7.3 Java class library for CICS basics

Using Java class library for CICS (JCICS) application programming classes, you can start CICS services from Java. To start a CICS service, simply call a Java method. The methods that JCICS provides use Java exception classes to return error messages to the caller. The exception handling mechanism is internally based on the CICS API response codes, which are set after execution of the CICS commands.

CICS resources, such as files, programs, temporary storage queues, and transient data queues, are represented by instances of the appropriate Java class. Therefore, to work with, for example, a temporary storage queue (TSQ), you first create an instance of a TSQ object. Then, assign it a name that corresponds to the name of the TSQ that you want to use, and use its methods to manipulate the queue:

```
TSQ tsq = new TSQ();
tsq.setName("MYTSQ");
tsq.writeItem("Hello World!").getBytes();
```

Often, you might want to initialize the JCICS objects that your program works with from within a constructor, and use the initialized objects from regular methods. In Example 7-1 (an excerpt from 7.8, “Using temporary storage queues” on page 215), we declare a TSQ to be a blank final instance variable. The `final` keyword indicates that the variable must be initialized in the constructor and cannot be reassigned after it is initialized.

**Example 7-1 Declaring a TSQ to be a blank final instance variable**

```java
public class Scratchpad {

    public final TSQ tsq;

    public Scratchpad(String tsqName) {
        this.tsq = new TSQ();
        tsq.setName(tsqName);
    }

    public void writeBytes(byte[] bytes) throws ItemErrorException, ... {
        try {
            tsq.rewriteItem(1, bytes);
        } catch (InvalidQueueIdException e) {
            tsq.writeItem(bytes);
        }
    }
}
```
The JCICS library structure and naming conventions are modeled to be fairly close to the traditional EXEC CICS programming interface. Therefore, translating the CICS calls in a traditional CICS program to JCICS calls in a Java application is fairly easy. The downside is that the JCICS way of doing things often does not quite “feel right” to a seasoned Java programmer. For example, to iterate over a set of related resources, CICS has the concept of browsing, which is similar to the Java idiom of using an iterator. However, the relevant JCICS classes do not implement the iterator interface; instead, they stand entirely on their own.

**Note:** All JCICS classes that represent CICS resources are designed to comply with the JavaBeans standard, so you can use them with a visual composition editor.

### 7.4 Input and output streams

Similar to any stand-alone Java application, JCICS applications can use the predefined streams: `System.in` to read input and `System.out` and `System.err` to print output.

Additionally, the Task class has two fields, `out` and `err`, which are directed at the user's terminal if the transaction is started from a terminal (in CICS parlance, if the principal facility associated with the transaction is a terminal). Otherwise, they are the same as `System.out` (for the out field) and `System.err` (for the err field).

However, the standard input stream (`System.in`) is never connected to the terminal. Rather, it is associated with a hierarchical file system (HFS) or zSeries file system (zFS) file whose name is set up in the JVM profile. In other words, it is not interactive in the sense that you can read user input from it. So, there is probably little use for the `System.in` stream, except maybe for reading configuration information at program startup.

`System.out` and `System.err` are connected to HFS or zFS files as well. Those files are created (or scratched) when the JVM starts.

The terminal might change each time that the JVM is reused, although `System.in`, `System.out`, and `System.err` remain connected to their appropriate HFS or zFS files.
7.5 Exception handling

Because anything that can go wrong eventually does go wrong, you must handle error conditions in your code.

As usual, error reporting and handling in JCICS is integrated into the standard Java exception handling mechanism.

In traditional languages, CICS indicates the success or failure of a CICS command by returning a condition code to your application program (sometimes called the RESP value because you use the RESP keyword to retrieve it). If everything went fine, the RESP value is NORMAL. If the RESP value is some value other than NORMAL, you can test the value and obtain what happened. Many RESP values also have an associated RESP2 value, which gives further details about the error. In Java, RESP codes are mapped to Java exception classes.
For each RESP value that can occur in CICS, there is one corresponding Java exception class. See Figure 7-3.

Figure 7-3  Part of the JCICS exception hierarchy

In Java, exceptions fall into two major categories: Checked exceptions and unchecked exceptions. When you call a method that might throw a checked exception, you are required to either handle the exception in your method or to declare your own method as throwing that exception. Unchecked exceptions,
however, need not necessarily be handled or declared because they usually represent conditions that an application program has difficulty recovering from. Therefore, it is impractical to be forced to handle them.

Example 7-2 shows a first example of exception handling in JCICS. The sample program tries to count the number of items in a temporary storage queue (we describe temporary storage queues in 7.8, “Using temporary storage queues” on page 215). Because there is no straightforward method in the CICS API, we must do this by reading the items in order until CICS responds with an error after trying to read past the last item.

**Example 7-2  Exception handling example, first version**

```java
package com.ibm.itso.sg258038;

import java.io.PrintWriter;
import com.ibm.cics.server.*;

public class ExceptionExample {

/** Count the number of items in a TS queue. */
private static int countItems(TSQ tsq) throws
InvalidRequestException,
    IOErrorException, LengthErrorException, InvalidSystemIdException,
    ISCInvalidRequestException, NotAuthorisedException,
    InvalidQueueIdException //
{
    int count = 0;
    ItemHolder item = new ItemHolder();
    try {
        while (true) {
            //
            (2)
            tsq.readNextItem(item);
            count++;
            //
            (3)
        }
        } catch (ItemErrorException e) {
            //
            (4)
            return count;
        }
    }
    
    public static void main(String[] args) {
```
TSQ tsq = new TSQ(); //
(5)
tsq.setName("MYTSQ");
PrintWriter out = Task.getTask().out;
try {
    out.println("Number of items in TSQ " + tsq.getName() //
(6)
    + ": " + countItems(tsq));
} catch (InvalidRequestException e) { //
(7)
    out.println("Invalid request");
} catch (IOException e) {
    out.println("I/O error");
} catch (LengthErrorException e) {
    out.println("Length error");
} catch (InvalidSystemIdException e) {
    out.println("Invalid system id");
} catch (ISCInvalidRequestException e) {
    out.println("Inter-system invalid request");
} catch (NotAuthorisedException e) {
    out.println("Not authorized to access queue");
} catch (InvalidQueueIdException e) {
    out.println("Invalid queue ID");
}
}

Notes on Example 7-2 on page 199:

- The countItems() method expects a parameter of type TSQ, and declares that it might throw several different exceptions, namely, all exceptions that can be thrown by starting the TSQ.readNextItem() method except for the one we handle ourselves.

- In a semi-infinite loop, we try to read the items in the temporary storage queue. The loop is not infinite because it is eventually terminated when no more items are left in the queue.

- At this point, the call to TSQ.readNextItem() succeeded, and we increment the number of items read.

- When there are no more items left in the queue, JCICS raises an ItemErrorException. The loop is terminated and control flow reaches the catch block. We return the number of items read so far.

- If any other exception occurred, however, we do not catch it in the countItems() method. Rather, the calling method must handle it, which is why
we listed all possible exception classes in the method declaration, except ItemErrorException, which is handled in the countItems() method itself.

- The main method creates a TSQ object and sets the queue name, and calls countItems() to print the number of items in the queue.

- Because the countItems() method declares several checked exceptions, we either must handle them or declare them to be thrown from the main() method. In Example 7-2 on page 199, we “handle” them by listing each possible exception in a catch block and printing a short error message corresponding to the exception type. Obviously, in production code we want to do much more to report the error properly and take action to deal with the problem.

We do not have to catch an ItemErrorException because it is already handled in the countItems() method. In fact, if we try to handle it, we get a compilation error.

If you take a closer look at the long list of catch clauses in the main method of Example 7-2 on page 199, you see that several exceptions cannot actually be thrown. For example, an InvalidSystemIdException or ISCInvalidRequestException can occur only when dealing with remote TS queues (that is, they live in a different CICS region). The queue that we use in this example, however, is local (we never started the setSysId() method).

Therefore, in a second version of the example, Example 7-3, we chose to specifically handle only the exceptions that are somewhat likely to occur and handle all others in a generic way, rendering them unexpected.

**Example 7-3  Exception handling example, second version**

```java
public static void main(String[] args) {
    TSQ tsq = new TSQ();
    tsq.setName("MYTSQ");
    createItems(tsq);
    PrintWriter out = Task.getTask().out;
    try {
        out.println("Number of items in TSQ " + tsq.getName() + ": " +
        countItems(tsq));
    } catch (NotAuthorisedException e) {
        out.println("Not authorized to access queue");  // (1)
    } catch (InvalidQueueIdException e) {
        out.println("Invalid queue ID");               // (2)
    } catch (CicsException e) {
        out.println("Unexpected CICS exception: " + e);  // (3)
    }
}
```
Notes on Example 7-3 on page 201:

In this version of the example, we chose to handle only NotAuthorisedException and InvalidQueueIdException. The catch clause in the example handles all other exceptions. Order is important: A compilation error is displayed if a catch clause for a more generic exception textually appears before a catch clause for a more specific one (that is, for one of its subclasses). All checked exceptions that the JCICS API throws are subclasses of CicsException, so this clause handles all of them except the two that we chose to handle specifically.

Exception handling in CICS (or generally in Java, for that matter) is seemingly easy, but still often done wrong. It is important that your code does “the right thing,” not only if everything works smoothly, but also in case of failure. In Example 7-4, our intention is to protect a shared resource against concurrent updates using the NameResource mechanism (more about NameResource in “Synchronization” on page 191). Basically, a NameResource is a global flag that indicates whether some resource is in use.

The code in Example 7-4 looks simple enough: Acquire the lock on the shared resource, perform the update, and release the lock. If an error occurs (either when trying to acquire or release the lock, or when updating), an error message is logged.

Example 7-4   Incorrect exception handling: Resources held

```java
    // INCORRECT EXCEPTION HANDLING

    private void doUpdate() throws CicsException {
        // ... code omitted
    }

    private void updateSharedData() throws ResourceUnavailableException,
                           LengthErrorException {
        NameResource lock = new NameResource();
        lock.setName("sg258038.LOCK");
        try {
            lock.enqueue(); // Lock the shared resource.
            doUpdate(); // Perform the update.
            lock.dequeue(); // Release the lock.
        } catch (CicsException e) {
            logError("Update failed: " + e);
        }
    }
```
However, there is one serious flaw in the code: What happens when the lock is acquired successfully, but the update failed? Control is passed to the catch block, the lock.dequeue() call is never executed, and the program still holds the lock. Obviously, that is a bad thing because other instances of the application might want to access the protected resource as well. This problem is easy enough to fix, you might say. Just release the lock in the catch block as well, as shown in Example 7-5.

**Example 7-5  Still incorrect exception handling**

```java
try {
    lock.enqueue(); // Lock the shared resource.
    doUpdate(); // Perform the update.
    lock.dequeue(); // Release the lock.
} catch (CicsException e) {
    logError("Update failed: " + e);
    lock.dequeue(); // Release the lock.
}
```

But that is not much better. First, we have duplicate code, and second, what if the call to logError() fails? We still have the same problem. Swapping the two lines in the catch block is not much better either because the dequeue() call might fail, and the error is never logged.

The correct way to do it is to use the `try/catch/finally` mechanism. Code in a `finally` block always gets control, no matter if the corresponding `try` block completed successfully or raised an exception. Example 7-6 shows how this is done.

**Example 7-6  Correct exception handling using try/catch/finally**

```java
private void doUpdate() throws CicsException {
    // ... code omitted
}

private void updateSharedData() throws ResourceUnavailableException,
LengthErrorException {
    NameResource lock = new NameResource();
    lock.setName("sg258038.LOCK");
    lock.enqueue(); // Lock the shared resource.
    try {
        doUpdate(); // Perform the update.
    } catch (CicsException e) {
        logError("Update failed: " + e);
    }
}
```
In Example 7-6 on page 203, the call to lock.dequeue() is in a finally block, and is therefore executed regardless if the doUpdate() call or the logError() call succeeded or failed, which ensures that the lock is released in any event. Also, we no longer have duplicate code.

7.6 Calling other programs and passing data

In traditional CICS programming, it is common to split an application into several programs. Each program handles a specific part of the application, for example, you can have one front-end program that handles user interaction and one or more subprograms that handle the business logic, such as retrieving data from and inserting data into a database or Virtual Storage Access Method (VSAM) file. The front-end program displays menus and data entry forms (maps in CICS parlance), possibly also for input verification, and calls one of the business logic subprograms, depending on the user's input.

To call another program, you use the CICS LINK mechanism. A LINK causes the starting program to be temporarily suspended and the linked-to program to be started. When the linked-to program returns, control is given back to the original program. The started program is still part of the same unit of work. Therefore, any work that is done to recoverable resources by both programs is backed out if the task does not complete successfully. In other words, a CICS LINK works just like a subroutine call except that the calling and called programs are not statically linked together.

To pass control to another program, use the XCTL (transfer control) command. The program issuing an XCTL does not receive control back when the target program terminates. Rather, the program that is up one level in the call hierarchy does.
See Figure 7-4 for an illustration of the program control flow with LINK and XCTL.

In most cases, the linking program must communicate with the linked-to program, for example, to pass in a function code that tells it what specific function to perform and to get results back from it. In CICS, this is done either by using the COMMAREA mechanism or by using the newer Container mechanism. A COMMAREA is simply a storage area that is owned by the linking program and is made available to the linked-to program. Containers are similar, but they offer advantages over COMMAREAs in certain circumstances, particularly if larger quantities of data must be passed.
In CICS Java applications, separation of logic into several distinct subprograms is not that common, primarily because the Java language lends itself well enough to separate different aspects of the application into different parts of the code, namely, into different classes. Calling separate programs using LINK or XCTL is also more expensive than internal calls between methods in Java programs.

In practice, however, you probably must call existing programs from your Java code, for example, if you want to access heritage modules that are too expensive to be reimplemented using Java.

7.6.1 Calling other programs by using LINK and XCTL

The Java equivalents to EXEC CICS LINK and EXEC CICS XCTL are the methods Program.link() and Program.xctl(), respectively. There are several versions of each method, taking different parameters, depending on whether and how you want to pass a COMMAREA to the called program. Therefore, to set up a LINK (or XCTL) to another program, you create an instance of class Program, call its setName() method to supply the name of the program you want to transfer control to, and call link() or xctl() as appropriate.

The usual pattern is to create and initialize the Program object in a constructor so that it can be used from your individual methods, as shown in Example 7-7.

Example 7-7 Typical usage of Program.link()

```java
public class LinkDemo {

    private final Program okk850;

    public LinkDemo() {
        okk850 = new Program();
        okk850.setName("OKK850");
    }

    public void callOKK850() throws InvalidRequestException, 
    LengthErrorException, InvalidSystemIdException, 
    NotAuthorisedException, InvalidProgramIdException, 
    RolledBackException, TerminalException {
        okk850.link();
    }

    ...
}
```
7.6.2 Passing data between programs

To pass data between a calling program and a called program, CICS programs typically use the COMMAREA mechanism. Since CICS TS V3.1, they have the alternative of using the Container mechanism. These two mechanisms have much in common, so we provide an overview of both first and then a more detailed discussion of each follows.

A COMMAREA is nothing more than an area of storage that is owned by the calling program and is made available to the callee. A COMMAREA is limited to a maximum of 32 kB of storage, though Containers do not have this restriction.

Containers are named blocks of data that are designed to pass information between programs. You can think of them as “named COMMAREAs”. Programs can pass any number of containers between each other and can pass a theoretically unlimited amount of data. Containers are grouped in sets called channels. A channel is analogous to a parameter list.

CICS does not care about the format or layout of the COMMAREA or Container, which is entirely up to the two programs that are involved. Typically, the format is defined in a host language data structure, such as a COBOL Level-01 record or a C struct.

In JCICS, there are six different varieties of the Program.link() method:

- Program.link()
  Performs a link without a COMMAREA.
- Program.link(byte[] commarea)
  Performs a link with a COMMAREA. The linked-to program can modify the COMMAREA.
- Program.link(byte[] commarea, int datalength)
  Performs a link with a COMMAREA, where only the first datalength bytes are passed to the linked-to program. The linked-to program can modify the entire COMMAREA, even if it is longer than datalength bytes.
- Program.link(Channel chan)
  Performs a link to the program that is passing a CHANNEL.
- Program.link(IByteBuffer commarea)
  Program.link(IByteBuffer in, IByteBuffer out)
  These methods using IByteBuffer were retained for compatibility only. They were supplied for integration with the IBM Record Framework library, which came with VisualAge for Java. Because the Record Framework was not
carried over to Rational Application Developer, they can be considered deprecated.

**Tip:** If you encapsulate the mapping of data fields to a byte array into a single Java class, with getters and setters for all of the fields, you can use this class in both the sending and receiving programs to reduce the risk of the two programs getting out of sync with each other. This is similar to using a shared copybook in communicating COBOL programs.


### 7.6.3 Communicating by using the COMMAREA

We demonstrate the COMMAREA mechanism with a simple program that just reverts the COMMAREA being passed, as shown in Example 7-8.

**Example 7-8  Program to revert the passed COMMAREA**

```java
package com.ibm.itso.sg258038;

import com.ibm.cics.server.CommAreaHolder;

public class Revert {

    public static void main(CommAreaHolder cah) {
        byte[] data = cah.value;
        int n = data.length - 1;
        for (int i = (n - 1) / 2; i >= 0; --i) {
            byte temp = data[i];
            data[i] = data[n - i];
            data[n - i] = temp;
        }
    }
}
```

To test the program, we write another little program, Example 7-9, which starts it by using Program.link().

**Example 7-9  Linking to the Revert program**

```java
public class RevertTest {

    private static String revert(String s) throws CicsException {
        Program revert = new Program();
```
revert.setName("REVERT");
byte[] bytes = s.getBytes();
revert.link(bytes);
return new String(bytes);
}

public static void main(CommAreaHolder cah) {
    PrintWriter out = Task.getTask().out;
    try {
        String original = "Hello World!";
        String reverted = revert(original);
        out.println();
        out.println("Original: " + original);
        out.println("Reverted: " + reverted);
    } catch (CicsException ex) {
        out.println("Oops: " + ex);
    }
}

Alternatively, you can use the CECI transaction to test it, as shown in Example 7-10.

Example 7-10  Testing the Revert program by using CECI

link program(REVERT) commarea('The quick brown fox')
STATUS:  COMMAND EXECUTION COMPLETE
EXEC CICS  LInk Program( 'REVERT ' )
    < Commarea( 'xof nwork kciuq ehT' ) < Length( +00019 ) > <
    Datalength() > >
    < SYSid() >
    < SYNconreturn >
    < Transid() >
    < INPUTMSG() < INPUTMSGLen() >>

RESPONSE: NORMAL
EIBRESP=+0000000000
EIBRESP2=+0000000000
PF 1 HELP 2 HEX 3 END 4 EIB 5 VAR 6 USER 7 SBH 8 SFH 9 MSG 10 SB 11 SF
7.6.4 Communicating through channels and containers

We demonstrate the use of channels and containers by changing the COMMAREA example program to use channels and containers instead of a COMMAREA, as shown in Example 7-11.

Example 7-11 Program to revert the data that is passed in a container

```java
public class Revert2 {
    public static void main(CommAreaHolder cah) {
        Task task = Task.getTask();
        Channel channel = task.getCurrentChannel(); // (1)
        if (channel != null) {
            try {
                Container record = channel.getContainer("Revert_Record"); // (2)
                byte[] data = record.get(); // (3)
                int n = data.length - 1;
                for (int i = (n - 1) / 2; i >= 0; --i) {
                    byte temp = data[i];
                    data[i] = data[n - i];
                    data[n - i] = temp;
                }
                record.put(data); // (4)
            } catch (ContainerErrorException e) { // (5)
                System.out.println("Container Error Exception "+
                "- probably container not found");
                e.printStackTrace();
            } catch (ChannelErrorException e) { // (5)
                System.out.println("Channel Error Exception "+
                "- probably invalid channel name or write to a read-only channel");
                e.printStackTrace();
            } catch (InvalidRequestException e) { // (5)
                System.out.println("CICS INVREQ condition "+
                " - Something has gone badly wrong!");
            }
        }
    }
}
```
There are several differences between this program and the program that uses a COMMAREA:

- We first must get the channel from the task, which allows us to get to the data that is passed to this program.

- After we have the channel, we can get the container that has the data that was passed to the program. In a more complex program, we can retrieve multiple containers from the channel and process each accordingly.

- As with a COMMAREA, we can get the data as a byte array and process it in the same manner.

- For simplicity, we reused the container to return the data to the calling program, but it is advisable to use a new container to return the data.

- There is a selection of exceptions that can be thrown when you work with containers. These exceptions map closely to the CICS return conditions that can occur when you use channels and containers in other languages. This program attempts to provide some indication of what the problem was that caused the exception, but it does not attempt to recover from the problem. In a more sophisticated program, you want to do more to recover from these exceptions. It is also possible to catch most or all of the exceptions with a more generic catch block, such as catch CICSException, but this reduces the opportunity to deal with different types of exceptions appropriately.

Once again, we can test the program with a small program, which is shown in Example 7-12. This example is similar to the previous one and also uses Program.link(), but this time it passes a container.

Example 7-12  Program link passing a container

```java
public class Revert2Test {
    private static String revert(String s) throws CicsException {
        e.printStackTrace();
    }
    catch (CCSIDErrorException e) {
        System.out.println("CICS CCSIDERR condition " +
        " - Code page conversion problem");
        e.printStackTrace();
    }
    else {
        System.out.println("There is no Current Channel");
    }
}
```
Program revert2 = new Program();
revert2.setName("REVERT2");
Task task = Task.getTask();
Channel data = task.createChannel("Revert_Data");  // (1)
Container record = data.createContainer("Revert_Record");  // (2)
record.put(s);   // (3)
revert2.link(data);  // (4)
byte[] bytes = record.get();
return new String(bytes);
}

public static void main(CommAreaHolder cah) {
  PrintWriter out = Task.getTask().out;
  try {
    String original = "Hello World!";
    String reverted = revert(original);
    out.println();
    out.println("Original: " + original);
    out.println("Reverted: " + reverted);
  } catch (CicsException ex) {
    out.println("Oops: " + ex);
  }
}

Again, there are differences between this program and the one using a COMMAREA:

- We create a channel, and give it a name.
- After we have a channel, we can create a container to hold the data that we want to pass to the program that is being called. A more complex program might create multiple containers on a single channel to be passed to a target program.
- The data can be placed in the container either as a byte array or as a string. If a string is passed to the container, it is automatically converted to a byte array by using the default encoding in the JVM.
- The call passes the reference to the channel that owns the container.

You can run this program in the same way that you ran the previous test program, and the output is the same.
7.6.5 COMMAREAAs versus channels and containers

Channels and containers offer many benefits over COMMAREAAs, especially where large (greater than 32 kB) quantities of data must be passed. A full discussion of these advantages, and how best to use them would fill an entire book, and fortunately, that book already exists in the form of the Redbooks publication, *CICS Transaction Server V3R1 Channels and Containers Revealed*, SG24-7227.

As a simple rule-of-thumb, the following rules can help:

- Where the application must pass large quantities of data, channels and containers are easily the best choice.
- For existing applications that use COMMAREAAs, and pass small quantities of data, the effort of converting the program might not be justified.
- For new applications, channels and containers are typically the default choice.

For the full benefit of channels and containers to be realized, effort must be applied in their correct usage in the application and in appropriate system configuration. *CICS Transaction Server V3R1 Channels and Containers Revealed*, SG24-7227 offers considerable guidance in both these areas.

7.7 Accessing remote resources

Most CICS resources, such as files, TS queues, and programs, need not actually be in the CICS region from which they are accessed. They can live in another region connected to it by using multiregion operation (MRO), intersystem communication (ISC), or Internet Protocol interconnectivity (IPIC) (see the CICS Intercommunication Guide, SC34-7172 for full details). Resources can either be defined as *remote*, in which case the program using the resource does not even know that the resource is owned by a different region, or a program can explicitly access a resource in another region.

In JCICS, resources that can be accessed remotely (in another region) are subclasses of the abstract class *RemoteableResource* (see Figure 7-5 on page 215). The RemoteableResource class has two methods, setSysId() and getSysId(), which are used to set and retrieve the name of the region that owns the resource. All of the JCICS resource classes, except WebService, extend RemoteableResource and so can be considered to be *removable*. 
Example 7-13 demonstrates how to start a new transaction in a different CICS region, using the StartRequest class, which we briefly described in 7.12, “Interval control” on page 236.

Example 7-13  Starting a transaction in another CICS region

```java
StartRequest startRequest = new StartRequest();
startRequest.setName("TRN2");  // Name of transaction to be started
try {
    startRequest.setSysId("PJA6");  // System ID of remote CICS region
    startRequest.issue();  // Issue request
} catch (CicsException e) {
    log("Error starting remote transaction: " + e);
}
```
7.8 Using temporary storage queues

Temporary storage queues (TS queues) are a simple mechanism for interprocess communication in CICS. A TS queue is an array of numbered slots or items. You can either add an item to a queue or replace an existing item that is
identified by its item number. The maximum number of items is 32767, and each item can hold up to 32763 bytes of data.

One more benefit of TS queues, other than being easy to use, is that they can be created dynamically. In other words, no additional setup is necessary in CICS to create a TS queue. If a queue did not exist before, it is automatically created when a new item is added to it. A TS queue can also be predefined if the queue is to have special characteristics, such as security, recoverability, or is remote.

A common use of a TS queue is as a scratchpad, which is a shared storage area that is available to all instances of an application running in the CICS region. In this section, we develop a simple Scratchpad class to demonstrate usage of TS queues with JCICS.

Example 7-14 shows a skeleton version of our Scratchpad class, just having a reference to the TS queue to use, and three constructors.

Example 7-14  Skeleton Scratchpad class

```java
package com.ibm.itso.sg258038;

// Import declarations omitted...

/**
 * Demonstrates how to use TS queues as a scratchpad.
 * @author Ulrich Gehlert
 */
public class Scratchpad {

    /** Default TS queue name. */
    public static final String DEFAULT_SCRATCHPAD_TSQ_NAME = "SCRATCH";// (1)

    /** The TS queue we are writing to. */
    private final TSQ tsq; // (2)

    /** Constructor using the default TS queue name. */
    public Scratchpad() { // (3)
        this(DEFAULT_SCRATCHPAD_TSQ_NAME);
    }

    /** Constructor using an explicit TS queue name. */
public Scratchpad(String tsqName) { //
(4)
    this(new TSQ());
    tsq.setName(tsqName);
}

/** Constructor using an explicit TS queue (which may be remote). */
public Scratchpad(TSQ tsq) {
(5)
    this.tsq = tsq;
}

Notes on Example 7-14 on page 216:

► This is the default queue name if the user of the class chose to use the
  no-argument constructor.

► The TS queue where the data goes to. The variable is declared final, so it
  must be initialized, either directly or indirectly, by each constructor.

► The no-argument constructor sets up a TS queue with the default name.

► Constructor taking an explicit TS queue name.

► Constructor taking an initialized TS queue object. Callers use this constructor
  if they want to write to a remote TS queue, for example, because they want
  the scratchpad to be available across CICS regions.

Next, we implement methods for writing data into and getting data from the
scratchpad, as shown in Example 7-15 and Example 7-16 on page 219.

Example 7-15  Writing data to the scratchpad

/**
 * Write a byte array into the scratchpad queue.
 *
 * @param bytes The byte array to be written.
 */
public void writeBytes(byte[] bytes) throws ItemErrorException,
    InvalidRequestException, IOException, LengthErrorException,
    InvalidSystemIdException, ISCInvalidRequestException,
    NotAuthorisedException, InvalidQueueIdException
{
    try {
        tsq.rewriteItem(1, bytes); //
(1)
    } catch (InvalidQueueIdException e) {
    }
// The queue didn't exist -- add item to queue, thus creating the queue
    tsq.writeItem(bytes); // (2)

/**
* Write a string to the scratchpad.
*
* @param s The string to be written to the scratchpad.
*/
public void writeString(String s) throws InvalidRequestException,
        IOErrorException, LengthErrorException,
        InvalidSystemIdException, ItemErrorException,
        ISCInvalidRequestException, NotAuthorisedException,
        InvalidQueueIdException
{
    writeBytes(s.getBytes()); // (3)
}

Notes on Example 7-15 on page 217:

- In Example 7-15 on page 217, we never write into any TS queue slots other than the first one. The slot numbers count from 1, not from 0.
- If we got an InvalidQueueIdException when trying to write to the TS queue, it did not exist. Therefore, we use the writeItem() method, which causes the queue to be created.
  - There is a possible race condition: Another task can create the queue right between our failed call to rewriteItem() and our call to writeItem(). We can solve the problem by protecting this section of code with a NameResource. For the sake of brevity, we ignored that possibility.
- This method, writeString, allows us to write strings into the queue by converting a string to a byte[] and writing that byte array to the queue. We implement a corresponding readString() method to read a string back. Users of the Scratchpad class must agree upon what kind of data to put on the scratchpad.
Example 7-16 shows an example of reading data from the scratchpad.

Example 7-16  Reading data from the scratchpad

/**
 * Read a byte array from the scratchpad.
 *
 * @return The byte array on the scratchpad, or <code>null</code> if
 * the
 *         scratchpad is empty.
 */
public byte[] readBytes() throws InvalidRequestException,
    IOErrorException, LengthErrorException, InvalidSystemIdException,
    ISCInvalidRequestException, NotAuthorisedException
Writing data {   
    try {
        ItemHolder item = new ItemHolder();
        tsq.readItem(1, item);
        return item.value;
    } catch (InvalidQueueIdException e) {
        return null;
    } catch (ItemErrorException e) {
        return null;
    }
}

/**
 * Read a string from the scratchpad.
 *
 * @return The string currently on the scratchpad.
 */
public String readString() throws InvalidRequestException,
    IOErrorException, LengthErrorException, InvalidSystemIdException,
    ISCInvalidRequestException, NotAuthorisedException,
    InvalidQueueIdException
{
    byte[] bytes = readBytes();
    if (bytes == null)
        return null;
    else
        return new String(bytes);
}
7.9 Performing serialization

When you want to ensure that a given resource cannot be accessed by more than one task at a time, you need some form of serialization mechanism.

A resource, in this context, can be a physical resource, such as a TS queue or a file. Or, it might be a virtual resource, such as a credit card number.

A well-known solution to ensure mutual exclusion is the semaphore mechanism, which in z/OS and CICS is referred to as ENQ/DEQ. A semaphore is an integer variable (a counter) with an associated queue. When a task issues an ENQ for a given resource, other tasks that try to ENQ on the resource are suspended until the first task releases the resource using a DEQ operation.

A resource in the context of this command is any string of one through 255 bytes, established by in-house standards, to protect against conflicting actions between tasks or to cause single threading within a program. Therefore, if you enqueue on the name of a file, another task attempting the enqueue() call on the same file name is unsuccessful, which does not stop some other task from using the file. It means that the enqueue request is unsuccessful if another task attempts an enqueue on the same character string, which in this case happens to be the same as the string of characters that make up the name of the file. If more than one enqueue() call is issued for the same resource by a given task, the resource remains owned by that task until the task issues a matching number of dequeue() calls, finishes its current unit of work, or terminates.
You can enqueue by string or by address. The JCICS classes provide an AddressResource class and a NameResource class, as shown in Figure 7-6.

The AddressResource class is supplied for compatibility with other languages that use an enqueue on address (or, in general, on any resource name that is binary). The address that is used must be passed to your Java program through a COMMAREA, TSQ, or other shared area. What you supply to the AddressResource class is a byte array.

The preferable enqueue method is the enqueue by (human-readable) name, that is, by string. From your Java program, you create a NameResource object and use the setName() method to supply a 1 – 255 character string. Then, issue the enqueue() method.

An enqueue is held only until the current unit of work finishes, that is, it is automatically released when you call Task.getTask().commit() or Task.getTask().rollback(). Also, it is released if a task ends abnormally.
In z/OS, each ENQ has a given scope that determines the visibility of the ENQ. An ENQ might be visible:

- At local scope, that is, only in the address space issuing it (for CICS applications, the CICS region).
- At system scope, that is, only in the z/OS image where the program is executing.
- At sysplex scope, that is, in the entire sysplex.

In CICS, an application program cannot by itself determine the scope of an ENQ. By default, each ENQ has local scope only. If you want to ensure serialization across multiple CICS regions, you must set up a special CICS resource called an ENQMODEL. For details, see CICS System Definition Guide, SC34-7185.

Restriction: The EXEC CICS ENQ command has an option that allows a task to hold an enqueue across unit-of-work boundaries (ENQ RESOURCE(...) TASK). Unfortunately, there is no corresponding option, or parameter, in the JCICS API.

Set up an ENQMODEL definition: This is so important that we say it again. An ENQ is local to the CICS region by default. To achieve serialization across multiple regions, possibly running on different z/OS images in the sysplex, you must set up an ENQMODEL definition.

7.10 Web, TCP/IP, and document services

CICS Web support is a collection of CICS services that support direct access to CICS application programs from web browsers. You can use CICS Web support with the following programs:

- Web-aware application programs that use the EXEC CICS WEB and EXEC CICS DOCUMENT application programming interfaces. See the CICS Application Programming Guide for more information.
- Programs that are designed to communicate with 3270 terminals using basic mapping support (BMS).
- Programs that are designed to be linked to from another program using a COMMAREA interface.
Applications can use the facilities of the CICS Web support to do the following functions:

- Map a URL to a request for CICS services.
- Interpret HTTP requests.
- Construct HTTP responses.
- Build HTML output for display by a web browser.

Although CICS Web support is designed primarily to provide communication between a web browser and CICS using HTTP, it also supports clients that send non-HTTP requests. The same components of CICS Web support are used to process HTTP and non-HTTP requests.

CICS has a facility that allows you to build up formatted data areas that are known as *documents*. Some examples of how these formatted areas, or documents, can be used, include the following actions:

- Constructing a COMMAREA
- Creating standard formats for printing (for example, using your own letterhead, address)
- Constructing HTML data to be displayed by a web browser

### 7.11 File control

By using the JCICS file control methods, you can query and manipulate *key-sequenced data set (KSDS)*, *entry-sequenced data set (ESDS)*, and *relative record data set (RRDS)* Virtual Storage Access Method (VSAM) files:

- **Key-sequenced data set (KSDS)**
  
  A key-sequenced data set has each of its records identified by a key. The key of each record is a field in a predefined position within the record. Each key must be unique in the data set.

- **Entry-sequenced data set (ESDS)**

  An entry-sequenced data set is one in which each record is identified by its relative byte address (RBA), that is, by the byte offset of its starting position within the file. Records are held in an ESDS in the order in which they were first loaded into the data set.

  New records added to an ESDS always go after the last record in the data set. You cannot delete records or alter their lengths. After a record is stored in an ESDS, its RBA remains constant. When browsing, records are retrieved in the order in which they were added to the data set.
Relative record data set (RRDS)

A relative record data set has records that are identified by their relative record number (RRN). The first record in the data set is RRN 1, the second is RRN 2, and so on.

Records in an RRDS can be fixed or variable length records, and how VSAM handles the data depends on whether the data set is a fixed or variable RRDS. A fixed RRDS has fixed-length slots that are predefined to VSAM into which records are stored. The length of a record on a fixed RRDS is always equal to the size of the slot. VSAM locates records in a fixed RRDS by multiplying the slot size by the RRN, which you supply on the file control request, to calculate the byte offset from the start of the data set.

A variable RRDS, however, can accept records of any length up to the maximum for the data set. In a variable, RRDS VSAM locates the records by using an index.

A fixed RRDS generally offers better performance. A variable RRDS offers greater function.
The operations that are available on files fall into the following categories, all of which are supported by the JCICS API (Figure 7-7):

- Adding new records
- Reading an existing record
- Deleting an existing record
- Changing an existing record
- Browsing the file

Figure 7-7  JCICS classes for file control

Here, we describe only KSDS file access. Accessing the other two types of files with JCICS is fairly similar.

In the rest of this section, we develop an implementation of the Java Map interface that is backed by a KSDS data set. Essentially, it is a kind of wrapper around the KSDS that allows manipulation of the data set in a way that looks more natural to Java programmers than raw JCICS API calls. This example is rather long and many of the implementation details have nothing to do with
JCICS in itself. However, we feel that it illustrates file access in JCICS nicely and can prove to be a useful starting point for your own code.

If you are familiar with Java collection classes, you know that a map is a collection of name-value pairs. You can associate a value with a key (put), delete an association (remove), retrieve the value associated with a key (get), and iterate over the map in three different ways (keys only, values only, or key-value pairs).

The implementation does not always fulfill the map contract. For example, the documentation for map says that the remove() method returns the value that is previously associated with the key or null if no value was previously associated with the key.

However, we chose not to return the previous value but just a Boolean that indicates there actually was a record in the file with the given key (or null if there was none) because of performance. If we want to return the previously associated value, we must perform an extra read operation. Because the value returned from remove() is not typically used anyway, we chose to consciously break the contract.

### 7.11.1 Class declaration and constructors

We start with the class declaration, which includes fields and constructors, as shown in Example 7-17.

```
Example 7-17  VsamMap class skeleton

package com.ibm.itso.sg258038.vsam;

import java.util.AbstractMap;
... more import statements ...

/**
 * An implementation of a Map that is backed by a KSDS VSAM file.
 * Note that, for simplicity and performance, this implementation violates some of the Map interface's contracts.
 * For example, the {link #remove(Object) remove()} method does not return the value previously associated with the key.
 * @author Ulrich Gehlert
 */
```
public class VsamMap extends AbstractMap { //
    private class VsamIterator implements Iterator { //
        ... see below ...
    }
    static class SimpleEntry implements Entry {
        ... not shown here (copied from java.util.AbstractMap)
    }
    /** Length of key. */
    private final int keylength;
    /** Underlying KSDS. */
    private final KSDS ksds; //
    private final RecordHolder recordHolder = new RecordHolder();
    /**
     * Constructor.
     */
    public VsamMap(KSDS ksds, int keylength) { //
        this.ksds = ksds;
        this.keylength = keylength;
    }
    /**
     * Convenience constructor that accepts a KSDS name.
     */
    public VsamMap(String ksdsName, int keylength) { //
        this(new KSDS(), keylength);
        ksds.setName(ksdsName);
    }
}

Notes on Example 7-17 on page 226:

- Rather than implement all methods of the Map interface ourselves, we extend the AbstractMap class, which already provides much of the implementation.
  In fact, the only abstract method in AbstractMap is entrySet(), so in theory, we need to only implement that. However, the default implementations of several other operations are not very efficient, so we override them (for example,
AbstractMap.get() iterates over all associations until it finds an entry with the given key.

- Eventually, this is an implementation of iterator, which iterates over the entries in the KSDS (see Example 7-23 on page 232).
- This is the object that represents the VSAM KSDS by which this map is backed.
- This constructor takes a KSDS and a key length. We need the key length later for iterating through the file.
- For convenience, this is a constructor that takes a String argument (the KSDS name) rather than a pre-initialized KSDS object.

### 7.11.2 Getting a record

Next, we implement the get() method, Example 7-18, which accepts a key and returns the associated value or `null` when no value is associated with the key (that is, if there is no record in the file having that key). Also, it is easy to implement containsKey() using get().

To get the value that is associated with the key, we try to read a record with the given key from the underlying VSAM KSDS.

*Example 7-18  VsamMap.get()

```java
public Object get(Object key) {
    try {
        ksd.read(keyToBytes(key), recordHolder);  // (1)
        return valueFromBytes(recordHolder.value);  // (2)
    } catch (RecordNotFoundException notfound) {
        return null;  // (3)
    } catch (CicsException e) {
        throw new RuntimeException(e);  // (4)
    }
}

public boolean containsKey(Object key) {
    return get(key) != null;  // (5)
}
```
Notes on Example 7-18 on page 228:

- Read a record with the given key from the KSDS. The first argument to KSDS.read() is the key, as a byte array. The second argument is a RecordHolder object, which contains the data after a successful read.

  Because, in general, our keys and records do not have the form of byte arrays, we call helper methods to convert keys and records to byte arrays, and vice versa. The default implementations (see Example 7-19) assume that both keys and records are string objects. Subclasses can override these methods to provide their own conversions.

- The KSDS read was successful. We return the record after converting it.

- When the KSDS did not contain a record with the given key, JCICS raises a RecordNotFoundException. Because the Map interface states that get() return null when no value is associated with the key, we catch the exception and do so.

- All other exceptions are wrapped into a RuntimeException. We cannot directly throw instances of CicsException because these are checked exceptions.

  Implementing containsKey() is straightforward. Check if get() returns non-null.

The next piece of code, in Example 7-19, provides default implementations of the conversion methods that we talked about in the notes on Example 7-18 on page 228.

Example 7-19  VsamMap: Conversion methods

```java
protected byte[] keyToBytes(Object key) {
    return ((String) key).getBytes(); // (1)
}

protected byte[] valueToBytes(Object value) {
    return ((String) value).getBytes();
}

protected Object keyFromBytes(byte[] bytes) {
    return new String(bytes); // (2)
}

protected Object valueFromBytes(byte[] bytes) {
    return new String(bytes);
}
```
Notes on Example 7-19 on page 229:

- This default implementation assumes that both keys and records are strings. Therefore, to convert the key to a byte array, we cast it to a string and convert that string to a byte array.
- To convert back, we construct a new string object from the byte array.

As mentioned before, these methods are default implementations only and are meant to be overridden by subclasses as appropriate (that is why they were declared protected).

### 7.11.3 Removing a record

Next, we implement remove(), which deletes a record from the underlying VSAM data set, as shown in Example 7-20.

**Example 7-20  VsamMap.remove()**

```java
public Object remove(Object key) {
    try {
        ksds.delete(convertToBytes(key)); //
        return Boolean.TRUE; //
    } catch (RecordNotFoundException e) {
        return null; //
    } catch (CicsException e) {
        throw new RuntimeException(e);
    }
}
```

Notes on Example 7-20:

- Again, we convert the key to a byte array, and then we call KSDS.delete().
- We return Boolean.TRUE if the record was deleted, which is a deliberate violation of the Map contract, which says that delete() returns the value previously associated with the key. Although fairly easy to implement, this causes another CICS call, which degrades performance.
- If the record was not found, return null.
7.11.4 Adding a record

To implement Map.put(), we try to write a record to the data set, as shown in Example 7-21.

Example 7-21  VsamMap.put()

```java
public Object put(Object key, Object value) {
  byte[] keyBytes = convertToBytes(key);
  byte[] valueBytes = convertToBytes(value);
  try {
    kds.write(keyBytes, valueBytes); // (1)
  } catch (DuplicateRecordException e) {
    throw new IllegalArgumentException("Duplicate key"); // (2)
  } catch (CicsException e) {
    throw new RuntimeException(e);
  }
  return null;
}
```

Notes on Example 7-21:
- Try to write the record to the KSDS.
- If we receive a DuplicateRecordException, we throw an IllegalArgumentException, which is explicitly permitted by the documentation of java.util.Map ("some aspect of this key or value prevents it from being stored in this map").

Therefore, effectively, this implementation only allows you to add new records but not to modify existing ones.

7.11.5 Iteration

Finally, we implement Map.entrySet(), which is the only abstract method in AbstractMap, as shown in Example 7-22, which adds support for iterating over the keys or values in the map, in other words, to browse the file.

Example 7-22  VsamMap.entrySet()

```java
public Set entrySet() {
  return new AbstractSet() { // (1)
    public Iterator iterator() {
      return null;
    }
  }
}
```
public int size() {  
    throw new UnsupportedOperationException();  // (3)
}
}

Notes on Example 7-22 on page 231:

- Again, we use an abstract implementation that is provided by the collections framework. The methods that we must implement are iterator() and size().
- See Example 7-23 for the implementation of VsamIterator.
- Because there is no easy way to obtain the number of records in a VSAM data set (other than counting), we chose not to support this operation.

The hardest part is implementing an iterator class, as shown in Example 7-23. It is implemented as an inner class of VsamMap and so has access to the VsamMap attributes.

Example 7-23  VsamMap.VsamIterator

```java
public class VsamIterator implements Iterator {
    private final KeyedFileBrowse fb;
    private final KeyHolder kh = new KeyHolder();
    private byte[] nextKey;
    private byte[] nextValue;
    private final RecordHolder rh = new RecordHolder();

    public VsamIterator() {
        try {
            fb = ksds.startBrowse(new byte[keylength]);  // (1)
        } catch (CicsException e) {
            throw new RuntimeException(e);
        }
    }

    public boolean hasNext() {  
        if (nextKey == null) {  // (2)
            try {
                //
            }
        }
    }

    public boolean hasNext() {  
        if (nextKey == null) {  // (2)
            try {
                //
            }
        }
    }
```
{ NEXT KEY VALUE = rh.VALUE; } CATCH (EndOfFileException e) { close(); // (4) CATCH (CicsException e) { close(); // (5) throw new RuntimeException(e); }
} return nextKey != null; // (6)

public Object next() {
   if (hasNext()) {
      Entry entry = new SimpleEntry(keyFromBytes(nextKey),
   valueFromBytes(nextValue));
   nextKey = nextValue = null;
   return entry;
   } else {
      throw new NoSuchElementException();
   }
}

public void remove() {
   throw new UnsupportedOperationException(); // (8)
}

public void close() {
   try {
      fb.end(); // (9)
   } catch (CicsException ignored) {
   }
}
}
Notes on Example 7-23 on page 232:

- In CICS terms, to move through a data set sequentially, you start a **BROWSE** operation. In JCICS, this is implemented by the startBrowse() method, which returns an instance of class KeyedFileBrowse. This class, in turn, has methods to return the next record from the file.
- To protect against multiple invocations of hasNext() without calling next() in between, we first check if the next record was already read.
- If not, we read it, using the next() method on the KeyedFileBrowse object that we received in the constructor call. We remember both key and value.
- If the end of file was reached, end the browse.
- If any other exception occurred, end the browse to free any resources held by it, and throw a RuntimeException.
- At this point, either the next record was successfully read, and nextKey (and nextValue) are **non-null**, or the end of file was reached, in which case nextKey remained **null**.
- The implementation of next() calls hasNext(). If there was another record, construct a key-value pair from it, and reset nextKey and nextValue to **null** so that hasNext() tries a physical read at the next invocation.
- There is no KeyedFileBrowse method to delete the current record, so we do not support this operation. It would be fairly easy to implement, however, by remembering the last key retrieved and calling KSDS.delete() with that key.
- This method ends the browse, cleaning up any resources held by it.

### 7.11.6 Testing the implementation

To test our VSAM-backed map, we run a small program, which is shown in Example 7-24.

Example 7-24  Testing the VsamMap implementation

```java
package com.ibm.itso.sg258038.vsam;

import java.util.Iterator;

public class VsamMapTest {
    private final VsamMap map;

    public static void main(String[] args) {
        try {
            new VsamMapTest();
        } catch (Exception e) {
            
```
private VsamMapTest() {
    String key = "66666";
    String record = createRecord(key);
    this.map = new VsamMap("ACCTFILE", key.length());
    System.out.println("Contains key "+key+": "+
    map.containsKey(key));
    testPutNew(key, record);
    System.out.println("Contains key "+key+": "+
    map.containsKey(key));
    testPutExisting(key, record);
    testRemove(key);
    printAllValues();
}

private void testPutNew(String key, String record) {
    System.out.println("Adding record " + key);
    map.put(key, record);
}

private void testPutExisting(String key, String record) {
    try { //
(1)
        System.out.print("Trying to modify record with key " + key + "...");
        map.put(key, record.toString());
        System.out.println("Oops, should have caused an exception");
    } catch (IllegalArgumentException expected) {
        System.out.println("Caught an IllegalArgumentException, as
        expected");
    }
}

private void testRemove(String key) {
    System.out.println("Removing record " + key);
    map.remove(key);
}

private String createRecord(String key) { //
(2)
    StringBuffer buf = new StringBuffer(key + "DOE                      JOHN ");
private void printAllValues() {
    Iterator iter = map.values().iterator();
    while (iter.hasNext()) {
        System.out.println(iter.next().toString().substring(0, 35));
    }
}

Notes on Example 7-24 on page 234:

- Modifying an existing record using VsamMap.put() is not allowed. Check that an IllegalArgumentException is thrown, as expected, which creates a record of the correct length for our example VSAM file, whose record length is 383 bytes. (The rest is padded with zeros.)
- Create a record.
- In each iteration, we print the first 35 bytes of the record.

Note: There is one potential flaw in our implementation: After you start iterating over the map, there is no way to end the iteration prematurely (to end the BROWSE operation, in CICS parlance), which can potentially tie up resources.

In particular, the number of concurrent operations that can be processed against a VSAM file is limited by the value of the STRINGS attribute on the file definition (for more information, see CICS Resource Definition Guide, SC34-7181). Therefore, when you open one iterator, and then at a later point open another one before the first one was at end-of-file, your program might block forever.

7.12 Interval control

An important facility in CICS is the ability to have one transaction start another transaction. You can have CICS start the transaction immediately (the default), at a specific time, or after a period. You can start the transaction on behalf of a specified terminal, and you can pass the started transaction some data. Additionally, the transaction can be started in another CICS region. The started
transaction is asynchronous to the transaction that starts it. Thus, the started transaction is in its own unit of work.

Optionally, you can pass data to the started transaction. Other than being able to retrieve the data passed to it, the started transaction is independent of the originating transaction.

We used this facility when we started the HelloWorld sample program using the START command from CECI. To do it from a JCICS program, you set up an instance of class StartRequest and call the issue() method.

Optionally, you can pass data to the transaction being started, similar to the COMMAREA mechanism explained in “Program control” on page 191. The difference is that because the starting transaction and the new transaction that is being started are independent from each other, the new transaction cannot pass data back to the originating transaction. Also, the mechanism to access the passed data is different: You use the Task.retrieve() method. Example 7-25 shows how it is done.

Example 7-25   Retrieving data that is passed from a START request

```java
package com.ibm.itso.sg258038;

import java.util.BitSet;
import com.ibm.cics.server.*;

public class RetrieveDemo {

    public static void main(CommAreaHolder cah) {
        RetrievedDataHolder rdh = new RetrievedDataHolder();
        BitSet whatToRetrieve = new BitSet(); //
        (1)
        whatToRetrieve.set(RetrieveBits.DATA);
        whatToRetrieve.set(RetrieveBits.QUEUE);
        try {
            Task.getTask().retrieve(whatToRetrieve, rdh);
            System.out.println("Retrieved DATA: " + new String(rdh.value.data)); //
            (2)
            System.out.println("Retrieved QUEUE: " + new String(rdh.value.queue)); //
            (3)
        } catch (EndOfDataException e) { //
            (4)
            System.err.println("No data to retrieve. Did you start me from a terminal?"));
    }
}
```
```java
} catch (InvalidRetrieveOptionException e) {
    System.err.println("Expected option not set by START command");
} catch (CicsException e) {
    e.printStackTrace();
}
```

Notes on Example 7-25 on page 237:

- We must indicate what to retrieve: Data, an RTRANSID, an RTERMID, a queue name, or some combination of these. (See CICS Application Programming Reference, SC34-6232, for more information about the meaning of the various options.)

- To indicate what combination to retrieve, we must instantiate a BitSet and set the corresponding bits (one of the constants in the RetrieveBits interface).

- Print the data and the queue name we received.

- No data was passed.

- We tried to retrieve a piece of information that was not present. That is, either no data or no queue name was passed from the transaction that started ours.

You can test the sample by using CEMT, as we show in Example 7-26.

Example 7-26   CEDA dialog to test Example 7-25 on page 237

```text
START TR(SMP1) From('Your data goes here') Queue(MYQUEUE)
STATUS: ABOUT TO EXECUTE COMMAND
EXEC CICS START
TRANSid( 'SMP1' )
  < Interval( +0000000 ) | TIME() | ( AFTER | AT ) < Hours() > < Minutes() >
  < SEconds() >
  < FROM('Your data goes here') < Length( +00019 ) < FMH > >
  < TERMid() | Userid() >
  < SYsid() >
  < RTRANSid() >
  < RTErmid() >
  < Queue( 'MYQUEUE ' ) >
  < Nocheck >
  < Protect >
  < REQid() >
  < ATTrach >
  < BRExit() < BRDATA() < BRDATALength() > >
```
You enter the data in the From clause of the START command (you must use quotations if the data contains embedded blanks) and the queue name in the Queue clause. Try to omit one or both of them, and inspect the program's output.

### 7.13 Terminal services

JCICS terminal services allow interaction with the user terminal. You can send data to and receive data from the terminal, and send cursor and device control instructions to the terminal.

Alas, using these services is not for the faint of heart because they require intimate knowledge of the 3270 data stream format (see 3270 Data Stream Programmer's Reference, GA23-0059).

There is one terminal service, however, that is relatively straightforward to use. If a JCICS program is started from a terminal, the Task.out variable (an instance of PrintWriter) allows you to write to that terminal, which is much like the standard Java System.out stream:

```java
Task.getTask().out.println("Hello World!");
```

That said, there is good news. During the residency, we developed a little package boldly called Java basic mapping support (JBMS), which provides an easy-to-use interface to the 3270 terminal, which is much like the corresponding basic mapping support (BMS) service in traditional CICS programming. Be aware, however, that the usual Redbooks publication disclaimer applies: It is experimental code, not thoroughly tested, and not ready for production purposes. It can come in handy, however, if you want to develop a simple front-end for an application without going through all of the effort to create a full-fledged web interface.
We show only some interesting parts of the JBMS package in the remainder of this section (see Figure 7-8).

![JBMS class hierarchy diagram]

**Figure 7-8  JBMS class hierarchy**
7.13.1 Classes

A brief explanation of all classes follows.

Component
This is the abstract superclass for all classes that represent elements on the window. All components except screen have a parent component that must be passed to the constructor. A component has a row and column position that is relative to its enclosing component (if any) and a flag that indicates whether the component is visible, that is, must be displayed on the window at all.

Container
This is an abstract superclass for all components that contain other components. The idea is that you add related items to a container using a relative position. So, if you later decide that you must move a container to another position on the window, you change the container’s position rather than the position of each individual item in the container.

Panel
Right now, a panel does not have more functionality over a container. The idea is that it can add visual elements, such as a border as an indication of grouping.

For convenience, we provide a subclass, PfKeyPanel, which you can use to show PF-key assignments.

Screen
This container represents an entire 3270 screen. It is responsible for generating and sending the outbound 3270 data stream to the terminal, and for receiving and parsing the inbound data stream.

Again, there is a convenience subclass, MenuScreen, which can be used to easily create a menu screen showing several items and an input field for the user’s choice.

Field
This is the abstract superclass of all basic components of a screen, that is, the actual fields that make up the screen.

Label
Represent fields that are intended for display only, that is, that are not for data entry by the user. However, the text displayed does not need to be static; it can be changed during the dialog with the user, for example, to show an error message.
Text
This is a field for data entry. Optionally, you can set a formatter, that is, an instance of java.text.Format to validate and format data entered by the user.

Next, we show you a sample panel created by using the JBMS package (Figure 7-9) and the program that created it (Example 7-27).

![Sample panel produced by a JBMS program](image)

**Figure 7-9** Sample panel produced by a JBMS program

Example 7-27 shows the program to generate this panel.

**Example 7-27 Sample JBMS program**

```java
package com.ibm.itso.sg258038;

import java.text.DecimalFormat;

import com.ibm.cics.server.*/;

import com.ibm.itso.sg258038.jbms.*;
import com.ibm.itso.sg258038.vsam.VsamMap;

public class BmsDemo {
    private Text acctno;
```
private Text firstname, lastname, midinit;
private Text phone;
private Text[] addr = new Text[3];
private PfKeyPanel pfkeys;
private VsamMap accounts = new VsamMap("ACCTFILE", 5); // (1)

public BmsDemo(TerminalPrincipalFacility term) throws
InvalidRequestException,
LengthErrorException, NotAllocatedException,
TerminalException
{
    Screen screen = createScreen();
    byte aid;
    String record = null;
    boolean end = false;

    term.clear();
    while (!end) {
        do {
            fillFields(record); // (2)
            aid = screen.displayAndReceive(term); // (3)
            } while (aid != AIDValue.PF3 && screen.validate() != null); // (4)
            switch (aid) {
                // ... omitted
            }
        }
        term.clear();
    }

    private String makeRecord() {
        // ... omitted
    }

    private void fillFields(String record) {
        // ... omitted
    }

    private Screen createScreen() { // (5)
        Screen screen = new Screen();
        Label label;
label = new Label(screen, 1, 1, "Account management");
label.setForegroundColor(Color.PINK);

label = new Label(screen, 4, 1, "Acct no:");
label.setForegroundColor(Color.TURQUOISE);
acctno = new Text(screen, 4, 10, 5); //
acctno.setFormat(new DecimalFormat("00000")); //

label = new Label(screen, 8, 1, "Phone:  ");
label.setForegroundColor(Color.TURQUOISE);
phone = new Text(screen, 8, 10, 10);

label = new Label(screen, 10, 1, "Address: ");
label.setForegroundColor(Color.TURQUOISE);
addr[0] = new Text(addressPanel, 0, 10, 24);
addr[1] = new Text(addressPanel, 1, 10, 24);
addr[2] = new Text(addressPanel, 2, 10, 24);

label = new Label(screen, 12, 1, "Name:   ");
label.setForegroundColor(Color.TURQUOISE);
lastname = new Text(namePanel, 0, 10, 18);
label = new Label(namePanel, 0, 29, ",");
label.setForegroundColor(Color.TURQUOISE);
firstname = new Text(namePanel, 0, 31, 12);
midinit = new Text(namePanel, 0, 44, 1);

label = new Label(screen, 14, 1, "PF:  ");
label.setForegroundColor(Color.TURQUOISE);

label = new Label(screen, 16, 1, "PF:  ");
label.setForegroundColor(Color.TURQUOISE);

pfkeys = new PfKeyPanel(screen);

return screen; //... (omitted)
public static void main(CommAreaHolder ca) {
    // ... omitted
}

Notes on Example 7-27 on page 242:

- The data displayed on this screen comes from a VSAM file. We use the VsamMap class developed in 7.11, “File control” on page 223 to access the data.
- Populate the fields with values from the record.
- Display the screen, and receive the user response. The fields that make up the screen are updated with the data that the user enters, and we get back an attention identifier (AID) value that tells us what key the user pressed to send the data (for example, the ENTER key, or a PF key).
- If the user entered invalid data (in this example, non-numeric data in the acctname field), start all over again, unless PF3 was pressed for exit.
- This method creates the screen, adding labels for informative text, and fields for data entry.
- This is the account number field. We set up a formatter that causes the data to be displayed with leading zeros and validates the user input.
- For easier maintenance, we group related fields, such as the name and address fields together, with offsets relative to the enclosing container. To move a group of related fields, change only the position of the container rather than of each contained element.
- Set the initial field, that is, the one that the cursor is positioned on when the screen is displayed.

7.13.2 Implementation

In the remainder of this section, we show you some implementation details. The most interesting parts are constructing the data to be sent to the 3270 terminal (in 3270 parlance, the *outbound data stream*), and parsing the data being sent back by the terminal (the *inbound data stream*).
Example 7-28 shows the method that creates the 3270 data stream and sends it to the terminal.

**Example 7-28  Screen.display()**

```java
public void display(TerminalPrincipalFacility term) throws
InvalidRequestException,
LengthErrorException, NotAllocatedException, TerminalException
{
    ByteArrayOutputStream bos = new ByteArrayOutputStream(); // (1)
    bos.write(0);
    bos.write(0);
    to3270(bos); // (2)
    if (getInitialField() != null) { // (3)
        // Position cursor to initial field.
        initialField.insertBufferAddress(bos, true);
        bos.write(Order.INSERT_CURSOR);
    }
    term.send(bos.toByteArray()); // (4)
}
```

Notes on Example 7-28:
- Create a `ByteArrayOutputStream` to hold the outbound 3270 data stream.
- Create the 3270 data stream, as shown in Example 7-29.
- If an initial field was specified, write a 3270 instruction to set the current buffer position and another one to move the cursor to that position.
- This is the JCICS call to send the data stream to the terminal.

To generate the data stream, we recursively iterate over all components, as shown in Example 7-29.

**Example 7-29  Container.to3270()**

```java
protected void to3270(ByteArrayOutputStream bos) {
    if (isVisible()) {
        // (1)
        for (Iterator iter = components.iterator(); iter.hasNext();) { // (2)
            Component component = (Component) iter.next();
```
component.to3270(bos); // (3)

Notes on Example 7-29 on page 246:

- If this container is visible at all:
  - Iterate over all contained components.
  - Tell each in turn to generate its part of the data stream, which is a recursive call if the current component is itself a container.

Finally, we must generate the 3270 stream for a single field, which is the most complicated part because here we finally must know about the 3270 data stream format.

Basically, the outbound 3270 data stream consists of command bytes (orders) followed by parameters. To define a field, use the Start Field (SF) or Start Field Extended (SFE) order. In our code, we use SFE.

A field begins at the current buffer position, which you can change with the set buffer address (SBA) order and ends right before the next field begins. SBA is followed by a two-byte buffer address.

Therefore, to start a new field, we first set the current buffer position, and then define the field using SFE and finally create an empty marker field.

A 3270 field can have several attributes, such as color, intensity, and whether the field is protected (no data can be entered into the field). Each attribute is recognized by an attribute type byte followed by an attribute value byte (see Table 7-2 on page 248). One of those attributes, the 3270 attribute, is always present, and is stored in the 3270 display buffer at the first position of the field (it does not display on the screen). The other attributes are optional.

The SFE order is followed by a single byte that indicates how many type-value byte pairs follow.
Table 7-2  Some 3270 attributes and their possible values

<table>
<thead>
<tr>
<th>Type code</th>
<th>Attribute type</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xC0</td>
<td>3270 Field attribute</td>
<td>0x20 Field is protected</td>
</tr>
<tr>
<td></td>
<td>Value is a combination of bits. Some combinations have special meanings (autoskip, nondisplay). Value must be EBCDIC encoded.</td>
<td>0x10 Field is numeric</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x30 Field is autoskip (no entry)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x08 Intensified</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x0C Nondisplay (password)</td>
</tr>
<tr>
<td>0x41</td>
<td>Extended highlighting</td>
<td>0x00 Default</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xF0 Normal (as determined by 3270 field attribute)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xF1 Blink</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xF2 Reverse video</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xF4 Underscore</td>
</tr>
<tr>
<td>0x42</td>
<td>Foreground color</td>
<td>0xF0 Neutral</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xF1 Blue</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xFF White</td>
</tr>
</tbody>
</table>

Example 7-30 shows how the data stream for a single field is generated.

Example 7-30  Field.to3270()

```java
protected void to3270(ByteArrayOutputStream bos) {
    if (isVisible()) {
        insertBufferAddress(bos, false);    // (1)
        bos.write(Order.START_FIELD_EXT);   // Start field extended    // (2)
        bos.write(countExtendedAttributes()); // Number of extended attributes    // (3)
        // Send extended attributes, as type/value byte pairs.    // (4)
        writeAttribute(bos, AttributeType.STD3270, (byte) Util.encode(attributes));
    }
}
```
writeAttributeConditional(bos, AttributeType.EXT_HILITE, extHilite);
writeAttributeConditional(bos, AttributeType.FOREGROUND, foregroundColor);
try {
    bos.write(text.getBytes());
} catch (IOException e) {
    // Can’t happen with a ByteArrayOutputStream
}
// Send a dummy marker field to indicate end of current field  
// (5)
bos.write(Order.START_FIELD);
bos.write(Util.encode(Attribute.AUTOSKIP));
}

Notes on Example 7-30 on page 248:

- First, we must tell the terminal the position of the field. The 3270 expects the position as a buffer address that is encoded in a rather peculiar format (see the full source code for details).
- Next, we indicate the start of a field using the Start Field Extended (SFE) order.
- SFE expects the number of attribute-value pairs.
- Write attribute-value pairs to the data stream.
- The 3270 assumes that a field ends where the next field starts, so we write a dummy marker field to indicate the end of the current one.

To get user input, we must receive and parse the inbound 3270 data stream, as shown in Example 7-31.

Example 7-31  Screen.receive() - Receive input from terminal, parse response, and update fields

/**
 * Receive input from terminal, and parse data stream.
 *
 * @return AID (Action Identifier) of the key pressed {one of the
 * @link com.ibm.cics.server.AIDValue AIDValue} constants).
 */
public byte receive(TerminalPrincipalFacility term) throws
   InvalidRequestException, LengthErrorException, NotAllocatedException,
   TerminalException
{
DataHolder dah = new DataHolder();
try {
    term.receive(dah); //
(1)
} catch (EndOfChainIndicatorException expected) {
    // Always thrown from TerminalPrincipalFacility.receive(). //
(2)
    // We can safely ignore it.
}
parseResponse(dah.value); //
(3)
return term.getAIDbyte(); //
(4)
}

Notes on Example 7-31 on page 249:

- JCICS call to receive the response from the terminal:
  Term.receive() always throws an EndOfChainIndicatorException (because the underlying EXEC CICS API always sets the corresponding RESP condition), and we can safely ignore it.

- Parse the response and return the AID that caused the screen to be sent.

The inbound 3270 consists of SBA orders that indicate which field is being sent next, followed by the actual data (that is, whatever the user entered into the field). So, we inspect the data stream, looking for SBA orders and finding the corresponding JBMS Field object for each order, as shown in Example 7-32. Then, we extract the field data and update the Field object accordingly.

Example 7-32  Parsing the response received from the 3270 terminal

private void parseResponse(byte[] resp) {
    // Start at index 1 (index 0 has a SET_BUFFER_ADDRESS for first field).
    for (int pos = 1; pos < resp.length; pos++) { //
(1)
        // Get buffer address.
        int bufaddr = ((resp[pos++] << 8) | (resp[pos++] & 0xFF)) & 0xFFFF;
        // (2)
        // Find end of current segment (look for next SBA).
        int end;
        for (end = pos; end < resp.length && resp[end] != Order.SET_BUFFER_ADDRESS; end++)
            ;
        // Find the field this segment corresponds to.
    }
Field field = findFieldByAddress(bufaddr); //

(3)
if (field != null) {
   // Found field; set its contents to the data received.
   String fieldContents = new String(resp, pos, end - pos); //

(4)
   field.setText(fieldContents);
}
pos = end;
}

Notes on Example 7-32 on page 250:
► Scan the data stream, and examine it for SBA orders.
► Extract the buffer address (next two bytes after SBA).
► Look up the field that this buffer address refers to.
► Extract the field data, and set the new field text.

Look to Chapter 8, “Interacting with structured data” on page 253 to describe JZOS. Also, see the JZOS Installation and User’s Guide, SA23-2245-00 and the book, Java Stand-alone Applications on z/OS Volume II, SG24-7291-00.
Interacting with structured data

Many existing CICS applications are centered around the use of structured data records, which are typically stored as sequential files in the Virtual Storage Access Method (VSAM) data set or as DB2 tables. The structured data is usually described using a COBOL copybook that conforms to COBOL levels, naming conventions, and data types. This copybook is then used as the interface to the data itself, because it defines the data types, and offsets within the data for the individual fields in the record.

When developing modern Java applications, it is frequently desirable to reuse the copybooks, which describe these existing record structures, to simplify the interactions from the Java application with the structured data required by the existing CICS applications.

This chapter provides an introduction into the IBM JZOS Batch Toolkit for z/OS software development kits (SDKs), which can be used to import record structures into a Java application. This chapter also provides an example of how we used the JZOS toolkit to create a list function for the GENAPP sample CICS COBOL application.
8.1 Overview

When developing Java applications, there is frequently a need to interact with structured data records that are used to describe input within the communications area (COMMAREA) to be passed on link calls between different programs (Figure 8-1). For example, existing adapter or presentation logic might be written in COBOL or PL/I and this might need to interact with new business logic components written in Java, which might then access record-based data in VSAM data sets or DB2 tables.

**GENAPP**

The GENAPP application is a sample CICS insurance policy application developed by the IBM CICS development team. GENAPP illustrates how a 3270-based *green screen* application can be transformed to use many of the rich features and functions of the newer releases of CICS Transaction Server (TS). It is designed to exercise various components of CICS, and provides a set of reusable functions to query, add and update policy information stored in a DB2 database. It is well structured into presentation and business logic components enabling different access methods to be easily developed.

In the following sections, we show you how to use the JZOS record importer to generate Java classes from the GENAPP COBOL copybooks. We then show you how to create the listing functions in Java by querying data from DB2 and then listing the information of the customers and policies in the web pages.

**Tip:** In addition to the JZOS record importer, IBM also provides the Java EE Connector Architecture (J2C) record importer in Rational Application Developer. This provides a graphical interface for building Java classes from COBOL, C, or PL/I-based copybooks. We show an example of how to use this tool in Appendix B, “Java EE Connector Architecture tools” on page 375.
8.2 JZOS toolkit

The IBM JZOS Batch Toolkit for z/OS SDKs is a set of tools designed to help integrate Java applications with existing services and interfaces on z/OS. One of the key features that it provides is a simple record class generator, which can be used with the high-level language COBOL or Assembler copybooks to generate Java records.

In addition, the JZOS toolkit also provides a native launcher for running Java applications directly as batch jobs or started tasks, and a set of Java wrapper classes that can access native z/OS application programming interfaces (APIs) directly from Java applications. Extra system services include console communication, multi-line write to operator (WTO), and return code passing capability. The JZOS APIs can be used in any z/OS environment including CICS applications, and are useful as a method to access partitioned data set (PDS) or zSeries file system (zFS) files directly from Java applications.

In the following sections, we describe how to use the JZOS toolkit with the GENAPP sample COBOL application to create Java records. We then describe how to use these records to develop Java components within the GENAPP applications.

8.2.1 Installing JZOS to z/OS

The runtime support for the JZOS APIs is provided by default in the IBM SDK for Java on z/OS. However, to run the record generator utility it is necessary to download and install the JZOS package from the IBM AlphaWorks website. For more information, see the JZOS Batch Launcher and Toolkit function in IBM SDK for z/OS Installation and User’s Guide, SA23-2245-04.

During installation, it is necessary to create a new zFS directory containing the supplied dynamic link libraries (DLLs) and Java ARchive (JAR) files. This directory is then referred to as <JZOS_HOME>. After installation, the following directories and data sets (Table 8-1) are created.

<table>
<thead>
<tr>
<th>Name</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>JZOS_HOME</td>
<td>/u/cicsrs5/jzos</td>
</tr>
<tr>
<td>Sample</td>
<td>CICSRS5.JZOS.SAMPJCL</td>
</tr>
<tr>
<td>Loadlib</td>
<td>CICSRS5.JZOS.LOADLIB</td>
</tr>
</tbody>
</table>
8.2.2 Running the record class generator

The following section describes the steps involved in running the JZOS record class generator:

1. To start the JZOS record class generator, it is first necessary to run the IBM Enterprise COBOL for z/OS compiler against the COBOL source with the ADATA option. This creates a SYSADATA file as output.

2. The second step is to run the com.ibm.jzos.recgen.cobol.RecordClassGenerator Java class by using the ADATA output from step 1, specifying the following arguments:
   - `adataFile=adataFileName` (default: //DD:SYSADATA)
   - `outputDir=dirName` (default: write .java file to System.out)
   - `symbol=RECORD-NAME` (default: first group data item found)
   - `packageName=fully.qualified.pkg.name` (default: “mypackage”)
   - `class=className` (default: the top-level record name)
   - `bufoffset=true|false` (default: false. See “bufOffset feature”)
   - `genSetters=true|false` (default: true)
   - `genCache=true|false` (default: true)
   - `stringTrim=true|false` (default: false)
   - `genAccessorJavadoc=true|false` (default: false)

If `outputDir` is given, the output file is created in the package-qualified subdirectory with a file name `className.java`. For example, if `outputDir=/u/myid`, `package=com.myco.cobrecs`, and `class=MyClass` are implicitly determined from the copybook record MY-RECORD, the output file created is: `/u/myid/com/myco/cobrecs/MyClass.java`.

If the given `adatafile` is a path name not starting with “/”, it is assumed to be the name of a binary file in the zFS (or workstation) file system that contains the binary SYSADATA file. To process SYSADATA files on the workstation, they must be downloaded with IBM record descriptor words (RDWs) included. This can be accomplished by using File Transfer Protocol (FTP) along with the `bin` and `quote site rd` commands.

If the given `adatafile` is a PDS name without a member or a directory name, all members of the PDS (or files in the directory) are processed. For this to work properly, the `outputDir` option must be specified without the class and symbol options so that the copybook top-level record name is used to generate unique class names for each class. This feature allows for one invocation of the RecordClassGenerator to generate Java classes for an entire PDS library of SYSADATA files.
The `RecordClassGenerator` writes the generated Java source file to `System.out`, so this file should be redirected to the file where you want to store your Java source. This would be something like: `src/com/myco/records/MyClass.java`, assuming that `packageName=com.myco.records` and `class=MyClass`.

The `genCache` parameter can be used to control whether code is generated to use instance variables to cache field values. Regardless of this setting, fields that are the objects (sizes) of OCCURS DEPENDING ON clauses are always cached.

The `stringTrim` parameter controls whether the generated code configures its `DataTypeFactory` to build trimmed `StringFields`.

The `genAccessorJavadoc` parameter can be used to cause comment Javadoc to be generated for field getter and setter methods to include the same COBOL source lines that are used to tag the static Field declarations. If set to false, no comments are generated for Field getter and setter methods.

**RecordClassGenerator example job**

The COBGEN job control language (JCL) shown in Example 8-1 can be used to run the COBOL compiler to create a `SYSADATA` file and then run the `RecordGlassGenerator` utility to build Java records for each file. Alternative JCL to run the High Level Assembler compiler can be found in the DSECTGEN sample JCL, provided by the IBM JZOS toolkit.

**Example 8-1   COBGEN sample JCL**
```plaintext
//XXXXXXXX JOB (),NAME
/*
***************************************************************************
//* Batch job to generate a Java class from a COBOL Copy Book
//* Tailor the proc and job for your installation:
//* 1.) Modify the Job card per your installation's requirements
//* 2.) Customize the JCL SET variables
//* 3.) edit JAVA_HOME to point to the location of the Java SDK
//* 4.) edit JZOSAW_HOME to point to the JZOS alphaWorks directory
//* 5.) Modify MAINARGS DD arguments to RecordClassGenerator
***************************************************************************
// SET COBPRFX='DDESHR.COB340'
// SET CPYBKLIB='JZOSAW.SAMPLE.JCL'
/*
//* Inline proc to compile a COBOL copy book to a temp ADATA PDS
//COBADATA PROC CPYBOOK=
//COBCC EXEC PROC=IFYWC,LNGPRFX=&COBPRFX,PARM='LIB,ADATA,MAP'
//COBOL.SYSLIB DD DISP=SHR,DSN=&CPYBKLIB
//COBOL.SYSADATA DD DSN=&&ADATA(&CPYBOOK),
```
// DISP=(MOD, PASS), SPACE=(CYL, (3, 3, 10))
// COBOL.SYSIN DD DSN=&CPYBKLIB(COBPREFX), DISP=SHR << COBOL PGM BEGIN
// DD DSN=&CPYBKLIB(&CPYBOOK), DISP=SHR << COPYBOOK
// DD DSN=&CPYBKLIB(COBSUFIX), DISP=SHR << COBOL PGM END
// PEND
//
// Compile one or many copy books to temp &ADATA PDS
// EXEC COBADATA, CPYBOOK=MEDICARE
// EXEC COBADATA, CPYBOOK=FIELDS
//
// Generate a .java file for each copy book
// JAVA EXEC PROC=EXJZOSVM, VERSION='50'
// MAINARGS DD *
com.ibm.jzos.recordgen.cobol.RecordClassGenerator
bufoffset=false
package=com.myco.cobol.records
outputDir="~/cobgen"
// SYSADATA DD DSN=&ADATA, DISP=(OLD, DELETE)
// STDENV DD *
# This is a shell script which configures variables
../etc/profile
###########################################################
# Customize below to match your installation:
# JAVA_HOME - The location of the SDK
# JZOSAW_HOME - The location of the JZOS alphaworks preview
###########################################################
export JAVA_HOME=<JAVA_HOME>
JZOSAW_HOME=<JZOSAW_HOME>
export PATH=/bin: "$JAVA_HOME"/bin:
LIBPATH=/lib:/usr/lib:"$JAVA_HOME"/bin
LIBPATH="$LIBPATH":"$JAVA_HOME"/bin/classic
LIBPATH="$LIBPATH":"$JZOSAW_HOME"
export LIBPATH="$LIBPATH":
# Force the alphaworks version of JZOS to run overriding the SDK
OPTS="-Djava.ext.dirs="$JZOSAW_HOME":"$JAVA_HOME"/lib/ext"
export JZOS_JVM_OPTIONS="$OPTS"
# Add the JZOS alphaworks jars to the classpath
for i in "$JZOSAW_HOME"/*.jar; do
CLASSPATH="$CLASSPATH": "$i"
done
export CLASSPATH="$CLASSPATH":
# Set JZOS specific options
# Use this variable to specify encoding for DD STDOUT and STDERR
#export JZOS_OUTPUT_ENCODING=Cp1047
# Use this variable to prevent JZOS from handling MVS operator commands
8.3 Using JZOS with the general insurance application

In this section, we describe how we used the JZOS record class generator with the general insurance application (GENAPP) COBOL application to develop a simple Java application for use with the GENAPP to list all the client and policy information for a specific customer.

**GENAPP**

The GENAPP is provided in SupportPac CB12, and demonstrates application modernization of CICS applications through the example of a general insurance application.

GENAPP is written in COBOL, and is designed to allow you to try out different features of CICS, including modernizing applications. It simulates transactions made by an insurance company to create and manage its customers and insurance policies. The application provides sample data and a 3270 interface for creating and inquiring on customers and policy information.

GENAPP stores client information and policy information in DB2. The database contains the following tables:

- A customer table that contains the customer records, including the customer number.
- A policy table that contains the policies, including the customer number, policy number, and policy type.
- A policy table for each type of policy: Commercial, endowment, house, and motor insurance policies.

GENAPP provides the following CICS transactions to create new policies:

**SSP1** Create a motor insurance policy
SSP2  Create an endowment insurance policy
SSP3  Create a house insurance policy
SSP4  Create a commercial property insurance policy
SSC1  Inquire on existing customer records and add customer records

For more information about GENAPP, see the SupportPac documentation that is available at the following website:


8.3.1 Generating Java classes for customer and policy records

To develop our new list function, we must understand the structured data that GENAPP uses to describe customer information and policy information. This information is contained within the copybook, LGPOLICY, in the DB2-CUSTOMER and DB2-POLICY records. See Example 8-2.

Example 8-2  Customer and policy records in the LGPOLICY copybook

```
... 01 DB2-CUSTOMER.
   03 DB2-FIRSTNAME   PIC X(10).
   03 DB2-LASTNAME    PIC X(20).
   03 DB2-DATEOFBIRTH PIC X(10).
   03 DB2-HOUSENAME   PIC X(20).
   03 DB2-HOUSENUMBER PIC X(4).
   03 DB2-POSTCODE    PIC X(8).

01 DB2-POLICY.
   03 DB2-POLICYTYPE   PIC X.
   03 DB2-POLICYNUMBER PIC 9(10).
   03 DB2-POLICY-COMMON.
      05 DB2-ISSUEDATE  PIC X(10).
      05 DB2-EXPIRYDATE PIC X(10).
      05 DB2-LASTCHANGED PIC X(26).
      05 DB2-BROKERID   PIC 9(10).
      05 DB2-BROKERSREF PIC X(10).
      05 DB2-PAYMENT    PIC 9(6).
...```

To generate the Java classes, take the following steps:

1. Copy the COMMAREA COBOL copybook, LGPOLICY, to a sample directory and edit the sample JCL COBGEN (refer to Example 8-1 on page 257) to generate Java record classes for the two COBOL copybooks.
2. Tailor the procedure and job for your installation:
   a. Modify the Job card according to your installation requirements.
   b. Customize the JCL SET variables.
   c. Edit JAVA_HOME to point to the location of the Java SDK.
   d. Edit JZOSAW_HOME to point to the JZOS home directory.
   e. Modify MAINARGS DD arguments to RecordClassGenerator, as shown in Example 8-3.

   **Note:** We show how to generate the Java class GenAppCustomer from the DB2-CUSTOMER record. It is also necessary to generate the GenAppPolicy class from the DB2-POLICY record using the same process.

   **Example 8-3 MAINARGS DD arguments for GenAppCustomer**
   
   ```
   //MAINARGS DD *
   com.ibm.jzos.recordgen.cobol.RecordClassGenerator
   bufoffset=false
   package=com.cics.genapp
   class=GenAppCustomer
   symbol=DB2-CUSTOMER
   outputDir=/u/cicsrs3
   ```

   Notes about Example 8-3:
   - **package** is the package name of the generated Java class
   - **class** is the class name of the generated Java class
   - **symbol** specifies the name of the level 01 element in the COBOL copybook
   - **outputDir** is the directory where we want to put the generated Java class
   - For further details about how to use the RecordClassGenerator, see 8.2, “JZOS toolkit” on page 255

3. Submit the JCL, which produces a Java class based on the provided copybook.

   **Note:** There is an alternative way to generate Java classes using the jzos_recgen.jar. This requires you to first generate the ADATA file using the COBOL compiler, and then use jzos_recgen.jar under zFS with the arguments described in 8.2, “JZOS toolkit” on page 255.
The JZOS generated Java classes

We now have two Java classes *GenAppCustomer* and *GenAppPolicy*. The GenAppCustomer class was generated from the DB2-CUSTOMER level 01 element in the COBOL program, and the GenAppPolicy class was generated from the DB2-POLICY level 01 element.

For each field in the copybook, there are a set of accessor methods to store data and a *get* method to read data. The Java classes use a fixed-length byte array to store all the data that can be retrieved by method *getByteBuffer()* and then used as a byte array in a COMMAREA or container to communicate with other programs. In Example 8-4 and Example 8-5 on page 264, we show the generated Java classes *GenAppCustomer* and *GenAppPolicy*.

**Example 8-4  Class GenAppCustomer**

```java
package com.ibm.cics.genapp;
import com.ibm.jzos.fields.*;

public class GenAppCustomer {
    protected static CobolDatatypeFactory factory = new CobolDatatypeFactory();
    static { factory.setStringTrimDefault(false); }

    /** <pre>
    01 DB2-CUSTOMER. </pre> */
    public static final int DB2_CUSTOMER_len = 72;

    /** <pre>
    03 DB2-FIRSTNAME PIC X(10). </pre> */
    protected static StringField DB2_FIRSTNAME = factory.getStringField(10);

    /** <pre>
    03 DB2-LASTNAME PIC X(20). </pre> */
    protected static StringField DB2_LASTNAME = factory.getStringField(20);

    /** <pre>
    03 DB2-DATEOFBIRTH PIC X(10). </pre> */
    protected static StringField DB2_DATEOFBIRTH = factory.getStringField(10);

    /** <pre>
    03 DB2-HOUSENAME PIC X(20). </pre> */
}```
protected static StringField DB2_HOUSENAME =
    factory.getStringField(20);

/** <pre>
  03 DB2-HOUSENUMBER          PIC X(4). </pre> */
protected static StringField DB2_HOUSENUMBER =
    factory.getStringField(4);

/** <pre>
  03 DB2-POSTCODE             PIC X(8). </pre> */
protected static StringField DB2_POSTCODE =
    factory.getStringField(8);

protected byte[] _byteBuffer;

public GenAppCustomer (byte[] buffer) {
    this._byteBuffer = buffer;
}

public GenAppCustomer () {
    this._byteBuffer = new byte[DB2_CUSTOMER_len];
}

public byte[] getByteBuffer() {
    return _byteBuffer;
}

public String getDb2Firstname() {
    return DB2_FIRSTNAME.getString(_byteBuffer);
}

public void setDb2Firstname(String db2Firstname) {
    DB2_FIRSTNAME.putString(db2Firstname, _byteBuffer);
}

public String getDb2Lastname() {
    return DB2_LASTNAME.getString(_byteBuffer);
}

public void setDb2Lastname(String db2Lastname) {
    DB2_LASTNAME.putString(db2Lastname, _byteBuffer);
}

public String getDb2Dateofbirth() {
    return DB2_DATEOFBIRTH.getString(_byteBuffer);
public void setDb2Dateofbirth(String db2Dateofbirth) {
    DB2_DATEOFBIRTH.putString(db2Dateofbirth, _byteBuffer);
}

custom String getDb2Housename() {
    return DB2_HOUSENAME.getString(_byteBuffer);
}

custom void setDb2Housename(String db2Housename) {
    DB2_HOUSENAME.putString(db2Housename, _byteBuffer);
}

custom String getDb2Housenumber() {
    return DB2_HOUSENUMBER.getString(_byteBuffer);
}

custom void setDb2Housenumber(String db2Housenumber) {
    DB2_HOUSENUMBER.putString(db2Housenumber, _byteBuffer);
}

custom String getDb2Postcode() {
    return DB2_POSTCODE.getString(_byteBuffer);
}

custom void setDb2Postcode(String db2Postcode) {
    DB2_POSTCODE.putString(db2Postcode, _byteBuffer);
}

Example 8-5 shows the generated Java class GenAppPolicy.

---

Example 8-5  Class GenAppPolicy

```java
package com.ibm.cics.genapp;
import com.ibm.jzos.fields.*;

public class GenAppPolicy {
    protected static CobolDatatypeFactory factory = new CobolDatatypeFactory();
    static { factory.setStringTrimDefault(false); }
    /** <pre>
     * 01 DB2-POLICY. </pre> */
    public static final int DB2_POLICY_len = 83;
```
/** <pre>
03 DB2-POLICYTYPE           PIC X. </pre> */
protected static StringField DB2_POLICYTYPE =
factory.getStringField(1);
/** <pre>
03 DB2-POLICYNUMBER         PIC 9(10). </pre> */
protected static ExternalDecimalAsLongField DB2_POLICYNUMBER =
factory.getExternalDecimalAsLongField(10, false, false, false, false);
/** <pre>
03 DB2-POLICY-COMMON. </pre> */
public static final int DB2_POLICY_COMMON_len = 72;
public static final int DB2_POLICY_COMMON_offset =
factory.getOffset();
/** <pre>
05 DB2-ISSUEDATE         PIC X(10). </pre> */
protected static StringField DB2_ISSUEDATE =
factory.getStringField(10);
/** <pre>
05 DB2-EXPIRYDATE        PIC X(10). </pre> */
protected static StringField DB2_EXPIRYDATE =
factory.getStringField(10);
/** <pre>
05 DB2-LASTCHANGED       PIC X(26). </pre> */
protected static StringField DB2_LASTCHANGED =
factory.getStringField(26);
/** <pre>
05 DB2-BROKERID          PIC 9(10). </pre> */
protected static ExternalDecimalAsLongField DB2_BROKERID =
factory.getExternalDecimalAsLongField(10, false, false, false, false);
/** <pre>
05 DB2-BROKERSREF        PIC X(10). </pre> */
protected static StringField DB2_BROKERSREF =
factory.getStringField(10);
/** <pre>
05 DB2-PAYMENT           PIC 9(6). </pre> */
protected static ExternalDecimalAsIntField DB2_PAYMENT =
factory.getExternalDecimalAsIntField(6, false, false, false, false);
protected byte[] _byteBuffer;

public GenAppPolicy (byte[] buffer) {
    this._byteBuffer = buffer;
}

public GenAppPolicy () {
    this._byteBuffer = new byte[DB2_POLICY_len];
public byte[] getByteBuffer() {
    return _byteBuffer;
}

public String getDb2Policytype() {
    return DB2_POLICYTYPE.getString(_byteBuffer);
}

public void setDb2Policytype(String db2Policytype) {
    DB2_POLICYTYPE.putString(db2Policytype, _byteBuffer);
}

public long getDb2Policynumber() {
    return DB2_POLICYNUMBER.getLong(_byteBuffer);
}

public void setDb2Policynumber(long db2Policynumber) {
    DB2_POLICYNUMBER.putLong(db2Policynumber, _byteBuffer);
}

public String getDb2Issuedate() {
    return DB2_ISSUEDATE.getString(_byteBuffer);
}

public void setDb2Issuedate(String db2Issuedate) {
    DB2_ISSUEDATE.putString(db2Issuedate, _byteBuffer);
}

public String getDb2Expirydate() {
    return DB2_EXPIRYDATE.getString(_byteBuffer);
}

public void setDb2Expirydate(String db2Expirydate) {
    DB2_EXPIRYDATE.putString(db2Expirydate, _byteBuffer);
}

public String getDb2Lastchanged() {
    return DB2_LASTCHANGED.getString(_byteBuffer);
}

public void setDb2Lastchanged(String db2Lastchanged) {
    DB2_LASTCHANGED.putString(db2Lastchanged, _byteBuffer);
}

public long getDb2Brokerid() {
    return DB2_BROKERID.getLong(_byteBuffer);
}

public void setDb2Brokerid(long db2Brokerid) {
    DB2_BROKERID.putLong(db2Brokerid, _byteBuffer);
}

public String getDb2Brokersref() {
    return DB2_BROKERSREF.getString(_byteBuffer);
}

public void setDb2Brokersref(String db2Brokersref) {
    DB2_BROKERSREF.putString(db2Brokersref, _byteBuffer);
8.3.2 Creating the GENAPP list class

After generating the Java classes to represent the structured data, we created the list function for GENAPP. We called this class *GenAppList*.

The key requirements for GenAppList are that it can connect to DB2 to list all the customer information and to inquire the policy information for a specific customer. GenAppList stores the information in GenAppCustomer and GenAppPolicy objects, the classes for which were generated from the LGPOLICY copybook.

To test the functions, we used a COBOL program to issue the request. The COBOL program calls GenAppList with a container called *CustomerID*. If the request data in the container has the integer value of zero, GenAppList uses method *listAllCustomer()* to query all the customer information in DB2. It then returns the customer information and customer total number with containers *CustomerInfo* and *CustomerCount*. The COBOL program then selects a customer and sends the customer ID in the *CustomerID* container. This time, GenAppList calls the listPolicy() method to obtain the policy information for the customer identified by the customerNumber and returns the policy information and policy total number with containers *PolicyInfo* and *PolicyCount*. We explain the logic based on the source code of GenAppList.

**Code analysis of Class GenAppList**

In Example 8-6, we show the main function of class GenAppList where GenAppList is started with a channel, and then call the methods based on the request container content.

*Example 8-6  GenAppList main method*

```java
public static void main(String[] args) throws
InvalidRequestException, ChannelErrorException, CCSIDErrorException,
CodePageErrorException, SQLException, ContainerErrorException
{
    task = Task.getTask();
    if (task == null) {
```
From the code used in Example 8-6 on page 267, you can see that we first find the running task, locate the current channel, and then get the container called `customerID`. After that, we retrieve the data from the container into a byte array. Because we know that the data is a customer ID number, we transfer the data into an integer. Then, we call `listAllCustomer()` or `listPolicy(customerID)` methods based on `customerID` value.

Next, take a closer look at the `listAllCustomer()` method (Example 8-7).

**Example 8-7  GenAppList listAllCustomer() method**

```java
public static void listAllCustomer() throws ContainerErrorException, ChannelErrorException, InvalidRequestException, CCSIDEErrorException, CodePageErrorException, SQLException{
    int customerCount = 0;
    GenAppCustomer customer = new GenAppCustomer();
    String customerData = "";

    // Connect to DB2 using JDBC driver
    String url = "jdbc:db2os390sqlj:".intern();
    Connection con;
    con = DriverManager.getConnection(url);
```
Statement select1 = con.createStatement();
ResultSet rs = select1.executeQuery("SELECT * FROM GENASA1.customer");

// Analyze each record
while (rs.next()) {
    customerCount++;
    int customerID = rs.getInt(1);
    String custID = Integer.toString(customerID);
    if (custID.length() == 1) {
        custID = "0" + custID;
    }
    customer.setDb2Firstname(rs.getString(2));
    customer.setDb2Lastname(rs.getString(3));
    customer.setDb2Dateofbirth(rs.getString(4));
    customer.setDb2Housename(rs.getString(5));
    customer.setDb2Housenumber(rs.getString(6));
    customer.setDb2Postcode(rs.getString(7));
    String custInfo = new String(customer.getByteBuffer());
    customerData = customerData + custID + custInfo;
} // end of while loop

// Put customer information into container CustInfo
CustInfo = channel.createContainer("CustomerInfo");
CustInfo.put(customerData); // put the data into the channel

// Put the number of customers into container CustomerCount
CustCount = channel.createContainer("CustomerCount");
String customerCountStr = Integer.toString(customerCount);
if (customerCountStr.length() == 1) {
    customerCountStr = "0" + customerCountStr;
}
CustCount.put(customerCountStr);

// Close the result set, the statement and the connection
rs.close();
select1.close();
con.close();
}

In the listAllCustomer() method, we first establish a connection to DB2 using the Java Database Connectivity (JDBC) driver. For more information about JDBC, see Chapter 9, “Accessing DB2 by using JDBC and SQLJ” on page 277.
Example 8-8 is a table called GENASA1.customer, which stores the customer information. The definition of this table is shown in Example 8-8.

**Example 8-8   GENASA1.customer table definition**

```sql
CREATE TABLE GENASA1.customer (  
customerNumber INTEGER NOT NULL GENERATED BY DEFAULT AS IDENTITY  
  (START WITH 1000001, INCREMENT BY 1, CACHE 20),  
  firstName CHAR(10),  
  lastName CHAR(20),  
  dateOfBirth DATE,  
  houseName CHAR(20),  
  houseNumber CHAR(4),  
  postcode CHAR(8),  
  PRIMARY KEY(customerNumber))  
CCSID EBCDIC  
IN GENASA1.GENATS01;
```

From the table, you can see that there are seven fields defined in the table. Among these fields, only `customerNumber` is not defined in the LGPOLICY copybook, which also means that the field is not included in the generated Java class. So we must get the `customerNumber` from the DB2 table and put it into the container without the help of the GenAppCustomer class.

Because `customerNumber` is an integer type, if we want to put the value into a CHAR container, we must transform it into a byte array or string. This allows simple manipulation of the data within the COBOL program. In this case, we changed the integer into a string, which can then be directly put into the CHAR container.

For other fields, we use GenAppCustomer to store each field with the set method and get all the data with method `getByteBuffer()`. There are several customer records in DB2, so we use string `customerData` to combine all the data together and put it into container `CustomerInfo` in one operation.

**Note:** It is necessary to put all the data into a container with one operation, because the JCICS container `put()` method does not currently support appending data to an existing container.

In the `listAllCustomer()` method, we use another container, `CustomerCount`, to store the customer record number so that it can help our COBOL test program analyze the data stored in `CustomerInfo`.

Example 8-9 on page 271 shows how to get policy information based on a customer ID number in the `listPolicy()` method.
Example 8-9  GenAppList listPolicy() method

```java
public static void listPolicy(int customerNum) throws SQLException,
InvalidRequestException, ContainerErrorException,
ChannelErrorException, CCSIDErrorException, CodePageErrorException{
    int policyCount = 0;
    String policydata = "";
    GenAppPolicy policy = new GenAppPolicy();

    // Connect to DB2 using JDBC driver
    String url = "jdbc:db2os390sqlj:".intern();
    Connection con;
    con = DriverManager.getConnection(url);
    Statement select1 = con.createStatement();
    String cmd = "SELECT * FROM GENASA1.policy where customerNumber = " +
    Integer.toString(customerNum);
    ResultSet rs = select1.executeQuery(cmd);

    // Analyze each record in the result set and update the policy
    while (rs.next()) {
        policyCount++;
        policy.setDb2Policynumber(rs.getLong(1));
        policy.setDb2Expirydate(rs.getString(2));
        policy.setDb2Policytype(rs.getString(3));
        policy.setDb2Lastchanged(rs.getString(4));
        policy.setDb2Brokerid(rs.getLong(5));
        policy.setDb2Brokersref(rs.getString(6));
        policy.setDb2Payment(rs.getInt(7));
        String policyInfo = new String(policy.getByteBuffer());
        policydata = policydata + policyInfo;
    }

    // Put policy information into container PolicyInfo
    PolicyInfo = channel.createContainer("PolicyInfo");
    PolicyInfo.put(policydata);

    // Put the policy count into container PolicyCount
    PolicyCount = channel.createContainer("PolicyCount");
    String polCountStr = Integer.toString(policyCount);
    if (polCountStr.length()== 1){
        polCountStr = "0" + polCountStr;
    }
    PolicyCount.put(polCountStr);

    // Close the result set, the statement and the connection
```

Chapter 8. Interacting with structured data  271
rs.close();
select1.close();
con.close();

The logic in `listPolicy()` is similar to `listAllCustomer()` except that we issue a different SQL command and get policy data from another DB2 table called `GENASA1.policy` (see Example 8-10).

Just as in `listAllCustomer()`, we use a while loop to analyze each record from the DB2 result set, and put the value of each field into the GenAppPolicy object with the set methods. The String policy data is used to combine all the data together and put into the container, `PolicyInfo`. In addition, we also count the number of policy records and return the count in container, `PolicyCount`.

Example 8-10  `GENASA1.policy` table definition

```sql
CREATE TABLE GENASA1.policy (
    policyNumber    INTEGER NOT NULL GENERATED BY DEFAULT AS IDENTITY
                      (START WITH 1000001, INCREMENT BY 1, CACHE 20),
    customerNumber  INTEGER NOT NULL,
    issueDate       DATE,
    expiryDate      DATE,
    policyType      CHAR(1),
    lastChanged     TIMESTAMP NOT NULL WITH DEFAULT,
    brokerId        INTEGER,
    brokersReference CHAR(10),
    payment         INTEGER,
    commission      SMALLINT,
    PRIMARY KEY(policyNumber),
    FOREIGN KEY(customerNumber)
               REFERENCES GENASA1.customer (customerNumber)
               ON DELETE CASCADE)

CCSID EBCDIC
IN GENASA1.GENATS02;
```

8.3.3 Packaging the sample into an OSGi bundle

After we finish all the coding work, it is necessary to deploy our Java application into the CICS JVM server environment.

Our GenAppCustomer and GenAppPolicy classes were both generated by the JZOS record generator, and import classes from the `com.ibm.jzos.fields` package. To ensure that these classes are available to the Open Services
Gateway initiative (OSGi) bundle class loader, we must add the jzos_recgen.jar file into the OSGi plug-in project build path in Eclipse. To do this, we transferred the JAR file from z/OS to our workstation using FTP in binary mode from the <JZOS_HOME> zFS location:

1. Create a lib directory in the plug-in project, and copy the JAR file into this location (Figure 8-2).

![Figure 8-2 Lib directory in plug-in project](image)

2. Next, we had to modify the plug-in project build.properties as follows:
   a. Open the MANIFEST.MF with the Plug-in Manifest Editor.
   b. Click the Runtime tab, and add the imported JAR file in the Classpath window (Figure 8-3).

![Figure 8-3 Configure plug-in class path](image)

After doing this step, you should then see that Bundle-ClassPath is added in the MANIFEST.MF file. This parameter is then used in the CICS JVM server run time to let the OSGi framework know where the JAR file is located.
3. Open the **Build** tab, and add the JAR file in the Extra Classpath Entries window (Figure 8-4). This adds the `jars.extra.classpath` statement to the `build.properties` file, meaning the JAR file that is specified by this parameter is built into the OSGi bundle.

![Figure 8-4 Configure extra class path entry](image)

After configuring the `build.properties` file, we can export the project as an OSGi bundle, include it into a CICS bundle, and then deploy the CICS bundle. See Chapter 4, “Deploying from CICS Explorer SDK” on page 71 for further information about this process.

### Wrapping JARs as plug-in projects

**Tip:** When non-OSGi library JAR files such as `zos_recgen.jar` are required by an application, it is better OSGi practice to create a library JAR file rather than edit the class path, as shown in Figure 8-4 on page 274. This can be achieved by using the **New → Plug-in from Existing JAR Archives** wizard.

The new OSGi bundle created in this way wraps the existing JAR file and exports the packages required by other dependent bundles. The wrapper bundle can then be used by other plug-ins in the Eclipse integrated development environment (IDE) by importing the required packages. The bundle can also be exported as either a CICS bundle or a middleware bundle into the JVM server OSGi framework, allowing the framework to verify the bundle dependencies ahead of time.

### 8.3.4 Testing our sample code

To verify the Java application, we wrote a COBOL test program. This sends a request with `00000000` in container `CustomerID`, and then analyzed the containers `CustomerInfo` and `CustomerCount` returned from the GenAppList. Then, the COBOL program selects one of the customers and sends the customer ID in the `CustomerID` container. This time, GenAppList returns the policy information and policy total number with containers `PolicyInfo` and `PolicyCount`. 
To validate our program, we used the JPDA debugger to attach to the JVM server and set several breakpoints in GenAppList. For more information about using and configuring JPDA with the CICS JVM server, see 5.5.2, “Java Platform Debugger Architecture” on page 148.

After our COBOL test program issues the first request to list all the customers, we saw that GenAppList stops at the first breakpoint, as shown in Figure 8-5.

Figure 8-5  The first breakpoint in GenAppList
For the first request, GenAppList queries the DB2 customer table and lists all the customer information. We set a breakpoint after the while loop in the `listAllCustomer()` method. We can use the variables window to check whether DB2 data is retrieved as expected, as shown in Figure 8-6.

Figure 8-6  Breakpoint at listAllCustomer() method
Chapter 9. Accessing DB2 by using JDBC and SQLJ

Java applications commonly need to access or store data in databases. This chapter introduces how to use Java Database Connectivity (JDBC) or Structured Query Language for Java (SQLJ) to access DB2 within a CICS Java application running in the Open Services Gateway initiative (OSGi) Java virtual machine (JVM) server environment.
9.1 DB2 drivers for JDBC and SQLJ

This section describes the different DB2 drivers available for use with JDBC and SQLJ.

9.1.1 Different driver types

To provide access to DB2 data from Java applications, DB2 provides a JDBC driver. According to the JDBC specification, there are four types of JDBC driver architectures:

**Type 1**
Drivers that implement the JDBC application programming interface (API) as a mapping to another data access API, usually Open Database Connectivity (ODBC), also known as a *JDBC-ODBC bridge*. These drivers are not suitable for a high-performance environment. The DB2 database system does not provide a type 1 driver.

**Type 2**
Drivers that are written partly in the Java programming language and partly in native code, also known as the *Native-API driver*. These drivers use a native (non Java) client library specific to the data source to which they connect. Their code portability is limited because of this native code. The native code can be written in C or other high performance languages and it communicates natively with the database. Type 2 drivers might provide higher performance.

Note: For Java applications running in CICS, the type 2 driver is always used because processing must be controlled by the native CICS-DB2 Attachment facility.

**Type 3**
Drivers that use a pure Java implementation and communicate with a data server using a data-server-independent protocol. The data server middleware then communicates the client’s requests to the database using the native API. The middleware isolates the Java client from databases, so no Java code must be changed when used with a new database. However, the extra layer might also make application less efficient. The DB2 database system does not provide a type 3 driver.
Type 4
Drivers that are pure Java and implement the network protocol for a specific data source. The difference between type 3 and type 4 drivers is that there is no middleware layer. The driver calls the database using a native API.

DB2 for z/OS supports the IBM Data Server Driver for JDBC and SQLJ (known as the JCC driver). This combines type 2 and type 4 JDBC implementations in a single driver. Two versions of the JCC driver are available: Version 3.5x is JDBC 3.0-compliant, and Version 4.x is compliant with JDBC 4.0 or later. The driver is available in two packages:

- db2jcc.jar for JDBC 3.0 and earlier support
- db2jcc4.jar for JDBC 4.0 or later, and JDBC 3.0 or earlier support

The DB2 Java ARchive (JAR) files are for use during application development and are required in both the application development environment and the CICS JVM run time. The SQL compressed files (sqlj4.zip or sqlj.zip) supply extra utilities for use during the SQL preparation process.

Tip: Given that the JDBC 4.0 level of functionality is compatible with an earlier version, it is suggested that db2jcc4.jar is used for JDBC 3.0 or 4.0 support.

9.1.2 Differences between JDBC and SQLJ

The SQLJ runtime environment consists of an SQLJ runtime library that is implemented in pure Java that provides the JDBC and SQLJ interfaces. The SQLJ runtime library calls the JCC driver to connect to the target database. In the following sections, we describe the differences between JDBC and SQLJ and the benefits of static versus dynamic SQL.

Dynamic SQL and static SQL
Usage of the SQL language has two models: dynamic and static. Dynamic SQL statements are prepared and assembled at run time. Conversely, with static SQL, the bind and prepare operations are done offline to work out the access path and to precompile the SQL statements. Then at execution time, the database uses the prepared access path, which can potentially provide improved performance over the dynamic model.

SQLJ and JDBC
JDBC uses dynamic SQL, so all SQL statements in JDBC are assembled at run time. In contrast, SQLJ uses static SQL, which enables you to embed static SQL statements into Java programs. SQLJ was initially developed by IBM, Oracle, and Tandem to complement the dynamic SQL JDBC model with a static SQL model.
In the JDBC case, you can improve the performance of the dynamic SQL by preparing the statement first. This gives the database the opportunity to compile and cache the statement, which can improve performance if the statement is called multiple times. For more information, see PreparedStatement in 9.2.4, “JDBC interfaces for executing SQL” on page 287.

SQLJ clauses are embedded into Java code. Each clause is preceded by the symbol #sql. The SQLJ statement in Example 9-1 assigns the surname of the employee with employee number 000010 to Java identifier empName.

Example 9-1  An SQLJ sample

```java
String empName;
...
#sql [ctxt]
   {SELECT LASTNAME INTO :empName FROM EMPLOYEE WHERE EMPNO='000010'};
```

The SQLJ translator translates an SQLJ source file into a standard Java source file along with an SQLJ serialized profile that includes information about the static SQL statements in the SQLJ source file. The translator converts SQLJ clauses to standard Java statements by replacing the embedded SQL statements with both Java classes, and calls to the SQLJ runtime library. This involves a connection to the database to work out the access paths ahead of time. DB2 provides the db2sqljcustomize command to customize the serialized profiles and bind DB2 packages to the database.

For applications that do not need to create SQL statements dynamically, you can use SQLJ, which provides better performance than JDBC. If there are also some SQL statements that must be created dynamically, you can use JDBC calls in an SQLJ program, because SQLJ and JDBC statements might coexist in the same Java application.

### 9.2 JDBC application programming

JDBC is the most popular programming method for accessing DB2 data in a Java application. We now describe how to use the JDBC interface.
9.2.1 Example of a simple JDBC application

A good way to understand JDBC is to review a typical JDBC program (Example 9-2) which demonstrates the basic elements that any JDBC application must perform.

Example 9-2  JDBCSample application

```java
class JDBCSample {
  public static void main(String[] args) {
    String url = "jdbc:default:connection";
    String empNo;
    String FirstName;
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    int nRecNum = 0;
    Task t = Task.getTask();
    t.out.println("***** JDBCSample begin *****");
    try {
      // Create the connection
      con = DriverManager.getConnection(url);
      t.out.println("    getConnection" + url + " Finished");

      // Commit changes manually
      con.setAutoCommit(false);

      // Create the Statement
      stmt = con.createStatement();

      // Execute a query and generate a ResultSet instance
      t.out.println("SELECT EMPNO,FIRSTNME FROM DSN8910.EMP WHERE WORKDEPT='A00'");
    }
  }
}
```


rs = stmt.executeQuery("SELECT EMPNO,FIRSTNME FROM DSN8910.EMP WHERE WORKDEPT='A00'");

// Print all of the employee numbers to standard output
while (rs.next() && nRecNum < 3) {
    empNo = rs.getString(1);
    FirstName = rs.getString(2);
    t.out.println("Employee number = " + empNo + "    FirstName = " + FirstName);
    nRecNum++;
}

// Commit the UOW to allow close of the connection
con.commit();
t.out.println("**** CICS JAVA JDBC EXAMPLE FINISHED *****");
}
catch (SQLException e) {  
    System.err.println("SQLException information");
    while (e != null) {
        t.out.println("Error msg: " + e.getMessage());
        t.out.println("SQLSTATE: " + e.getSQLState());
        t.out.println("Error code: " + e.getErrorCode());
        e.printStackTrace();
        e = e.getNextException();
    }
    try {
        con.rollback();
    } catch (SQLException e2) {
        e2.printStackTrace();
    }
} finally {  
    // Close the ResultSet
    if (rs != null) {
        try {
            rs.close();
        } catch (SQLException e) {
        }
    }
    // Close the Statement
    if (stmt != null) {
        try {
            stmt.close();
        } catch (SQLException e) {
        }
    }
}
In the sample program in Example 9-2 on page 281, the following key steps are necessary to use JDBC:

1. This statement imports the `com.ibm.cics.server.Task` class, which contains the JCICS API for manipulating a CICS task.
2. These statements import the classes in the `java.sql` package, which contain the JDBC API that we will be using.
3. The database is identified by a database Uniform Resource Locator (URL). For details, see section 9.2.3, “Connecting to a database” on page 284.
4. Connect to the database. The `DriverManager` class is used to get a connection to the database using the database URL from step 3. The database connection is managed by the CICS DB2 attachment facility.

   **Attention:** In a CICS OSGi JVM server, it is not necessary to load the JDBC driver by using the `Class.forName()` method because this is performed by the OSGi bundle activator in the JDBC JAR file when the JVM server is started.

5. If autocommit is on, the database manager performs a commit operation after every SQL statement completes. Here, we set it to `false` because we control the commit operation separately in step 9.
6. Create a SQL `Statement` object.
7. The data retrieved from a table using a SELECT statement is returned in a `ResultSet` object.
8. Retrieve the first column value from the result set.
9. Commit the unit of work (UOW) by starting the `commit()` method on the `Connection` object. The DB2 JDBC driver converts these calls into a CICS sync point, resulting in the commit of the whole CICS UOW. For more
information, see 9.2.5, “Transaction control in JDBC applications” on page 289.

10. Error handling for JDBC applications is done by using try/catch/finally blocks. The catch of an SQLException ensures that all DB2 related errors can be handled in a single clause.

11. Start the `rollback()` method on the connection if an SQLException was thrown, to ensure that the UOW is rolled back in error scenarios.

12. The `finally` clause ensures that the result set, statement, and connection are closed, irrespective of whether any exceptions were thrown.

The following section provides a brief introduction on each of the key functions necessary to program with JDBC methods. For more details, review *DB2 for z/OS Application Programming Guide and Reference for Java*.

### 9.2.2 Java packages for JDBC support

To start JDBC methods, your application might need to use classes from several different Java packages. Depending on the program, you might need to import some or all of the following packages:

- `java.sql`: Contains the core JDBC API.
- `javax.sql`: Contains extensions to JDBC such as for creating DataSource objects, PooledConnection, XAConnection, and RowSet support.
- `com.ibm.db2.jcc`: Contains the implementation of the DB2 JCC driver.
- `javax.naming`: Contains classes and interfaces for the Java Naming and Directory Interface (JNDI), which are necessary if you use a managed DataSource.

### 9.2.3 Connecting to a database

DriverManager and DataSource are two different classes that a Java program can use to connect to a database via JDBC:

- The DataSource interface is more portable and is primarily designed for managed application server environments where the data source is created ahead of time and published to a naming service such as JNDI.
- The DriverManager interface is a simpler but less portable means of connecting to a database.
When you connect to a database by either connection method, you create a Connection object, which represents the connection to the database. You can then use this connection to do the following things:

- Create Statement, PreparedStatement, and CallableStatement objects for executing SQL statements.
- Gather information about the database to which you are connected.
- Commit or rollback transactions.
- Close the connection to the database.

**Using DriverManager**

Use the following method to establish a connection with the DriverManager:

- Connect to a database by starting the `DriverManager.getConnection()` method. There are three forms of `getConnection()`:
  - `getConnection(String url)`;
  - `getConnection(String url, user, password)`;
  - `getConnection(String url, java.util.Properties info)`;

When you use JDBC in CICS, the database connection is managed by the CICS DB2 attachment facility, which means that a user ID and password are not required in the Java code. The user ID is specified instead on the DB2CONN or DB2ENTRY resource definitions.

The JDBC driver recognizes two types of URLs, **default** and **explicit**.

**Default URL**

In a default URL, there is no local DB2 subsystem name. The JDBC driver will connect to local DB2 to which the CICS region is connected. A default URL does not include the location name of a DB2 subsystem. Therefore, if you are using the DB2 group attach facility, you can connect to any of your DB2 subsystems, not just one named subsystem.

A default URL for DB2 for z/OS can be specified in one of two formats:

- `jdbc:db2os390sqlij`: or `jdbc:db2os390`: For compatibility with older versions
- `jdbc:default:connection`: The recommended format for use in CICS with a local DB2 subsystem

**Explicit URL**

An explicit URL includes the location name of a DB2 subsystem. The basic structure of an explicit URL for DB2 for z/OS can be specified in one of two formats:

- `jdbc:db2os390:<location-name>`: For compatibility with old versions
- `jdbc:db2os390sqlj:<location-name>`: The new format

It is recommended that you use the default URL connection in a CICS environment, unless you need to connect to a remote DB2 using DB2 distributed data facility (DDF). If you use an explicit URL, you must issue an explicit commit before you can close the connection; this commits the entire CICS UOW. For more information about CICS UOW coordination with JDBC, see 9.2.5, “Transaction control in JDBC applications” on page 289.

**Using DataSource**

DataSource is another interface that provides the ability to connect to a database. If you use the DataSource method, instead of specifying a default or explicit URL in the program itself, an application can use the Java Naming and Directory Interface (JNDI) to look up a reference to a previously deployed DataSource definition.

Typically, an application server like WebSphere Application Server creates and maintains a pool of DataSources separately from the applications that use the connections. The DataSource definition provides all the necessary information to generate the JDBC connection, so the application programmer does not need to know the database URL. This makes your applications more portable.

In CICS, the JDBC driver converts the JDBC or SQLJ requests into their EXEC SQL equivalents, then flows them into the CICS DB2 attachment facility. This is the same process as for any EXEC SQL call made from a COBOL program. The CICS DB2 attachment facility has its own mechanism to manage, protect, and reuse threads. The JDBC connection pooling mechanism is superfluous in the CICS DB2 environment because CICS provides the equivalent function.

However, this does not preclude the use of the DataSource interface in a CICS Java application. The application can either use the DB2SimpleDataSource
interface to create its own data source reference, or a publishing application can be used to create and manage the DataSource using the JNDI.

**Note:** If using JNDI to look up a DataSource in a CICS TS V4.2 OSGi JVM server, you need to enable the JNDI implementation classes to register themselves with the OSGi framework. Further details about how to do this are detailed in the following IBM technoid:


CICS provides three sample Java applications to demonstrate how to use DataSources in CICS:

- CICSDatasourcePublish.java: Publish a CICS-compatible DataSource to a JNDI Namespace.
- CICSDatasourceRetract.java: Retract a CICS-compatible DataSource from a JNDI Namespace.
- CICSjdbcDataSource.java: Look up a CICS-compatible DataSource from a JNDI namespace, and use it to obtain a JDBC connection with default URL characteristics to the local DB2 subsystem to which CICS is connected.

You can find these examples in the /usr/lpp/cicsts42/samples/jdbc/ directory.

### 9.2.4 JDBC interfaces for executing SQL

To retrieve data from tables, update data in tables, or call stored procedures, you start JDBC methods. There are two choices of interfaces that you can use:

- **Statement:** Supports all SQL statement execution. There are two interfaces inherited from the Statement interface:
  - PreparedStatement: Supports SQL statements containing input variables.
  - CallableStatement: Supports the invocation of a stored procedure.
- **ResultSet:** Provides access to the results that a query generates. It is similar to the cursor that is used in SQL applications written in other languages.

**Note:** Using a prepared statement, particularly more than once, gives the driver and database a chance to optimize the call by working out and caching database access paths. However, because it must do this at run time, this is not as performant as SQLJ, which does the statement preparation offline.
Retrieve data from tables
To retrieve data from a table, you can use the `Statement.executeQuery()` method. This method returns a result table in a `ResultSet` object. After obtaining the result table, you use `ResultSet` methods to move through the result table and obtain the individual column values from each row.

Example 9-3 shows how to execute a SQL statement and display retrieved information.

Example 9-3   A SQL execution statement

```java
String empNo;
String firstName;
Connection con;
Statement stmt;
ResultSet rs;
...
stmt = con.createStatement();
rs = stmt.executeQuery("SELECT EMPNO,FIRSTNME FROM DSN8910.EMP WHERE WORKDEPT='A00'");
while (rs.next()) {
    empNo = rs.getString(1);
    firstName = rs.getString(2);
    System.out.println("Employee number = " + empNo + " First name = " + firstName);
}
```

To retrieve data from a table with parameter markers, you use the `PreparedStatement.executeQuery()` method. This method also returns a result table in a `ResultSet` object. The following code in Example 9-4 demonstrates how to retrieve rows from the employee table for a specific employee.

Example 9-4   Retrieve rows from table

```java
String empnum, phonenum;
Connection con;
PreparedStatement pstmt;
ResultSet rs;
...
pstmt = con.prepareStatement("SELECT EMPNO, PHONENO FROM EMPLOYEE WHERE EMPNO=?");
pstmt.setString(1, "000010");
rs = pstmt.executeQuery();
while (rs.next()) {
    empnum = rs.getString(1);
    phonenum = rs.getString(2);
    System.out.println("Employee number = " + empnum + " Phone number = " + phonenum);
}
```
Creating and modifying database objects

To create and update tables, the Statement interface provides the `executeUpdate()` method. The following operations are valid:

- Execute data definition statements, such as CREATE, ALTER, DROP, GRANT, and REVOKE.
- Execute INSERT, UPDATE, DELETE, and MERGE statements that do not contain parameter markers.
- Execute the CALL statement to call stored procedures. These procedures to CREATE or MODIFY tables have no parameters and return no result sets.

Example 9-5 shows an example of the `executeUpdate()` method.

For those SQL statements with parameter markers, `PreparedStatement` is the preferred interface to use rather than `Statement` and this also inherits the `executeUpdate()` method.

**Example 9-5  Statement.executeUpdate() method**

```java
Connection con;
Statement stmt;
int numUpd;
...
stmt = con.createStatement();
umUpd = stmt.executeUpdate("EXEC SQL INSERT INTO T1 VALUES (1, CURRENT DATE, 'TEST')");
stmt.close();
```

9.2.5 Transaction control in JDBC applications

In a JDBC application, you can select transaction isolation levels just as you can in other programming languages. JDBC isolation levels can be set using the `setTransactionIsolation()` method on the connection.
Table 9-1 shows the values that you can specify in the `setTransactionIsolation()` method and their DB2 database isolation level equivalents.

**Table 9-1  JDBC values for different isolation levels**

<table>
<thead>
<tr>
<th>JDBC value</th>
<th>DB2 isolation level</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.sql.Connection.TRANSACTION_SERIALIZABLE</td>
<td>Repeatable read</td>
</tr>
<tr>
<td>java.sql.Connection.TRANSACTION_REPEATABLE_READ</td>
<td>Read stability</td>
</tr>
<tr>
<td>java.sql.Connection.TRANSACTION_READ_COMMITTED</td>
<td>Cursor stability</td>
</tr>
<tr>
<td>java.sql.Connection.TRANSACTION_READ_UNCOMMITTED</td>
<td>Uncommitted read</td>
</tr>
</tbody>
</table>

To commit a unit of work, your JDBC and SQLJ applications issue commit and rollback method calls. In CICS, the DB2 JDBC driver converts these calls into a CICS sync point or rollback call. Therefore, a JDBC or SQLJ commit results in the whole CICS unit of work being committed, not just the updates made to DB2. The autocommit property also causes a CICS commit, and results in the whole unit of work being committed.

When you are using explicit URLs to connect databases, you are required to issue an explicit commit before you can close the connection. If this program is linked to by others and they are in same unit of work, this action commits all the work, which might be not desirable.

You can avoid the issue by acquiring the connection using a default URL instead of an explicit URL, or by using a DataSource that provides a default URL connection. When a default URL is used, the Java application does not have to commit work when it closes the connection. It is recommended that you always use default URL connections in a CICS environment.

### 9.2.6 Exceptions and warnings handling

In JDBC applications, SQL errors throw exceptions, which you handle by using try/catch/finally blocks. SQL warnings do not throw exceptions, so you need to start methods to check whether warnings occurred after you execute SQL statements.
SQLException
All JDBC methods throw an instance of SQLException when an error occurs during execution. An SQLException object contains the following information:

- An int value that contains an error code (SQLCODE).
  SQLException.getErrorCode() retrieves this value.
- A String object that contains the SQLSTATE, or null.
  SQLException.getSQLState() retrieves this value.
- A String object that contains a description of the error, or null.
  SQLException.getMessage() retrieves this value.
- A pointer to the next SQLException, or null.
  SQLException.getNextException() retrieves this value.

Example 9-6 shows code to handle an SQLException.

Example 9-6  A sample of handling SQLException

```java
try {
    con = DriverManager.getConnection (url);
    stmt = con.createStatement();
    ...
    rs = stmt.executeQuery("SELECT EMPNO,FIRSTNME FROM DSN8910.EMP WHERE WORKDEPT='A00'");
    ...
} catch(SQLException ex) {
    while(ex!=null) {
        t.out.println("Error msg: " + ex.getMessage());
        t.out.println("SQLSTATE: " + ex.getSQLState());
        t.out.println("Error code: " + ex.getErrorCode());
        ex.printStackTrace();
        ex = ex.getNextException();
    }
}
```

SQLWarning
The IBM Data Server Driver for JDBC and SQLJ accumulates warnings when SQL statements return positive SQLCODEs, and when SQL statements return 0 SQLCODEs with non-zero SQLSTATEs.
Calling `getWarnings()` retrieves an `SQLWarning` object.

A generic `SQLWarning` object contains the following information:

- A `String` object that contains a description of the warning, or null
- A `String` object that contains the SQLSTATE, or null
- An int value that contains an error code
- A pointer to the next `SQLWarning`, or null

### 9.3 SQLJ application programming

SQLJ is an alternative database access method. In general, it can provide better performance than JDBC because of the ability to use static SQL, which removes many of the overheads of dynamic SQL as used in JDBC.

#### 9.3.1 Example of a simple SQLJ application

Following is a sample SQLJ program (Example 9-7).

```java
Example 9-7  SimpleSQLJ.sqlj application
```

```java
package com.ibm.cicsjava;

import com.ibm.cics.server.Task;
import java.sql.*;
import sqlj.runtime.ExecutionContext;  

#sql context SqljCntxt1;  
#sql iterator NamedIterTables (String empno, String firstnme);  

public class SQLJSample {

    public static void main(String[] args) {

        SqljCntxt1 sqljCntx = null;
        NamedIterTables tablesIterator = null;  
        String workDept = "A00";  
        Task t = Task.getTask();
        t.out.println();
        t.out.println("****  SQLJ example begin  *****");

        try {
            try {
                // Establish Connection;
                sqljCntx = new SqljCntxt1("jdbc:default:connection", false);  

                t.out.println("****  SQLJ example begin  *****");

            } catch (Exception e) {
                System.out.println("Error while establishing connection: " + e.getMessage());
            }

        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }

    }

}
```

292   CICS and the JVM server: Developing and Deploying Java applications
t.out.println(" successful connect; autocommit is false");

// Limit the number of rows to return from ResultSets
ExecutionContext ec = new ExecutionContext(); 7
ec.setMaxRows(3);

// SELECT from an existing table;
t.out.println("SELECT EMPNO,FIRSTNME FROM DSN8910.EMP WHERE
WORKDEPT=" + workDept);
#sql [sqljCntx, ec] tablesIterator = { SELECT EMPNO,FIRSTNME
FROM DSN8910.EMP WHERE WORKDEPT=:workDept }; 8
t.out.println(" successful execution of SELECT");

// Step through Iterator
//t.out.println ("About to fetch from Iterator");
while (tablesIterator.next()) {
  t.out.println(" EmployNumber: <" +
tablesIterator.empno() + "> FirstName: <"+
  + tablesIterator.firstnme() + ">");
}
t.out.println(" successful processing of Iterator");

// close Iterator and Statement
tablesIterator.close(); 9
t.out.println(" successful close of iterator");
sqljCntx.close(); 10

} catch (SQLException e) { 11
  t.out.println();
t.out.println(" ***** ERROR - SQLException: " + e + ",
SQLState = " + e.getSQLState() + ", errorCode = "
  + e.getErrorCode());
e.printStackTrace();
}
t.out.println();
t.out.println("**** SQLJ example complete *****");
Notes on Example 9-7 on page 292:

1. Include sqlj.runtime package.

2. SQLJ connection context declaration clause. When the statement of #sql context SqljCntxt1 is translated, an SqljCntxt1 class is created. This is then used to define a database context, which is then used in step 4.

3. Iterator declaration clause. An iterator contains the result table from a query. SQLJ generates an iterator class for each iterator.

4. When SQLJ translates the statement #sql iterator NamedIterTables, a NamedIterTables class is created. You can use it to define an iterator.

5. workDept is a Java String object that is used as an SQLJ host identifier in an SQL statement at step 8.

6. Acquire a connection to the database by using the connection context and set autocommit to false to allow CICS to control UOW syncpointing.

7. The ExecutionContext class controls or monitors the execution of SQL statements.

8. Execute SQL statement by using the defined context and ExecutionContext. The results are returned into the tablesIterator.


11. SQL error handling.
9.3.2 SQLJ program preparation

Unlike a JDBC program, an SQLJ program needs preparation before running. The preparation involves translating, compiling, customizing, and binding programs. Figure 9-1 shows the steps to prepare an SQLJ program.

Take the following basic steps for SQLJ program preparation:

1. Run the SQLJ command from the z/OS UNIX System Services command line to translate and compile the source code. The SQLJ command generates a Java source program, optionally compiles the Java source program, and produces serialized profiles.

2. Run the db2sqljcustomize command from the z/OS UNIX System Services command line to customize the serialized profiles and bind DB2 packages.

9.3.3 Connecting to a database by using SQLJ

There are several different methods available for connecting to the database in an SQLJ application. It is possible to use either the JDBC DataSource or the DriverManager interfaces as described in 9.2.3, “Connecting to a database” on
In addition, an SQLJ session context can be used, either explicitly created or by using the default context.

Example 9-8 shows a code snippet of using an explicitly created session context to connect to a data source.

**Example 9-8 Sample SQLJ code to connect data source**

```java
#sql context SqljCntxt1;

SqljCntxt1 sqljCntx = null;
try {
    // Establish Connection;
    sqljCntx = new SqljCntxt1("jdbc:default:connection", false);
}
```

### 9.3.4 Variables in SQLJ applications

In SQLJ programs, you can use host variables to pass data between the Java application and DB2. You define Java variables in your application, use them in your SQL statements, leading by a colon (:). The colon is followed by an optional parameter mode identifier (IN, OUT, or INOUT), as shown in Example 9-9. Host variables are case-sensitive.

**Example 9-9 Use host variables in SQLJ**

```java
String hvmgr="000010";
String hvdeptno="A00";
#sql [ctx] {UPDATE DEPARTMENT SET MGRNO=:hvmgr WHERE DEPTNO=:hvdeptno};
```

### 9.3.5 SQL statement execution in SQLJ applications

In SQLJ applications, you run SQL statements by using SQLJ executable clauses. An executable clause can have one of the following general forms:

- `#sql [connection-context] {sql-statement}`
- `#sql [connection-context,execution-context] {sql-statement}`
- `#sql [execution-context] {sql-statement}`
connection-context:
Specifies a valid Java identifier that is declared earlier in the SQLJ program. That identifier must be declared as an instance of the connection context class that SQLJ generates for a connection declaration clause.

execution-context:
Specifies a valid Java identifier that is declared earlier in the SQLJ program.

Retrieving data in SQLJ applications
SQLJ applications use a result set iterator to retrieve result sets, similar to a cursor in other language. If you want to retrieve a single row from a table, you can select the columns into host variables, as shown in Example 9-10.

Example 9-10  Retrieve data in SQLJ

```sql
String custname;
#sql [Ctx] {SELECT CUSTNAME INTO :custname FROM CUSTOMER WHERE CUSTID="123456"};
```

In most cases, multiple rows might be selected from the table. In other languages, you declare a cursor, open it, and fetch each row from the result table. The equivalent in Java is a result set iterator. A result set iterator is a Java object that you use to retrieve rows from a result table. Unlike a cursor, a result set iterator can be passed as a parameter to a method.

Take the following basic steps to use a result set iterator:

1. Declare the iterator, which results in an iterator class
2. Define an instance of the iterator class
3. Assign the result table of a SELECT to an instance of the iterator
4. Retrieve rows
5. Close the iterator

There are two types of iterators: Positioned iterators and named iterators. Positioned iterators identify the columns of a result table by their position in the result table. Named iterators identify the columns of the result table by result table column names. The following two examples show how to use the different iterators.
Example 9-11 shows a named iterator example.

**Example 9-11  Named iterator**

```java
#sql iterator ByName(String LastName, Date HireDate);
   // Declare named iterator ByName
   {
       ...
       ByName nameiter;
       #sql [ctxt]
       nameiter={SELECT LASTNAME, HIREDATE FROM EMPLOYEE};
       while (nameiter.next())
       {
           System.out.println( nameiter.LastName() + " was hired on "
                               + nameiter.HireDate());
       }
       nameiter.close();
   }
```

Example 9-12 shows a positioned iterator example.

**Example 9-12  Positioned iterator**

```java
#sql iterator ByPos(String, Date);
   {
       ...
       ByPos positer;
       String name = null;
       Date hrdate;
       #sql [ctxt] positer =
               {SELECT LASTNAME, HIREDATE FROM EMPLOYEE};
       #sql {FETCH :positer INTO :name, :hrdate};
       while (!positer.endFetch())
       {
           System.out.println(name + " was hired in " + hrdate);
           #sql {FETCH :positer INTO :name, :hrdate};
       }
       positer.close();
   }
```
Modifying database objects in an SQLJ application

To perform a simple insert or update, if the SQL statement is determinate, you can execute the statement directly in SQLJ, as shown in Example 9-13.

Example 9-13  Execute statement directly

```sql
#sql [myConnCtx]
{INSERT INTO DEPARTMENT VALUES ("X00","Operations 2","000030","E01",NULL)};
```

In most cases, an update is performed during the browse of a result set. In such a situation, positioned UPDATEs or DELETEs should be used. Do the following basic steps to perform this function:

1. Declare the iterator.
   The iterator can be positioned or named. Declare the iterator as updatable.

2. Disable autocommit mode for the connection.
   If autocommit is performed, the iterator is destroyed (unless it has the holdability=true attribute set) during the autocommit.

3. Create an instance of the iterator class.

4. Assign the result table of a SELECT to an instance of the iterator.

5. Retrieve and update rows.
   Execute a FETCH statement to obtain the current row. If the iterator is pointing to a row to be updated, execute an SQL UPDATE... WHERE CURRENT OF :iterator-object statement to update the columns in the current row.

6. Close the iterator.

Example 9-14 and Example 9-15 on page 300 show a skeleton of a program to update recorders in a cursor.

First, in one file, declare positioned iterator UpdByPos, specifying that you want to use the iterator to update column SALARY.

Example 9-14  Declaring a positioned iterator

```sql
import java.math.*;
#sql public iterator UpdByPos implements sqlj.runtime.ForUpdate 1
   with(updateColumns="SALARY") (String, BigDecimal);
```
Example 9-15 shows a program structure on using cursor.

Example 9-15  Program structure on using cursor

```
import UpdByPos;
#sql context HSCtx;
public static void main (String args[])
{
  ...
  Connection HSjdbccon=
    DriverManager.getConnection("jdbc:default:connection");
  // Set autocommit off so automatic commits
  // do not destroy the cursor between updates
  HSjdbccon.setAutoCommit(false);

  // Create a connection context object
  HSCtx myConnCtx=new HSCtx(HSjdbccon);
  UpdByPos upditer; // Declare iterator object of UpdByPos class
  String empnum; // Declares host variable to receive EMPNO
  BigDecimal sal; // and SALARY column values
  #sql [myConnCtx] upditer = {SELECT EMPNO, SALARY FROM EMPLOYEE
    WHERE WORKDEPT='D11'};
  // Assign result table to iterator object
  #sql {FETCH :upditer INTO :empnum,:sal};
  // Move cursor to next row
  while (!upditer.endFetch())
  {
    #sql [myConnCtx] {UPDATE EMPLOYEE SET SALARY=SALARY*1.2
      WHERE CURRENT OF :upditer};
    // Perform positioned update
    System.out.println("Updating row for " + empnum);
    #sql {FETCH :upditer INTO :empnum,:sal};
    // Move cursor to next row
  }
  upditer.close(); // Close the iterator
  #sql [myConnCtx] {COMMIT};
  myConnCtx.close();
}
9.4 Deploying the JDBC and SQLJ samples

In this section, we describe how to deploy and run a JDBC and SQLJ sample application into a CICS JVM server environment.

9.4.1 CICS region setup

This section describes how to set up a CICS region to support deployment of JDBC applications into a JVM server.

DB2 setup

- Ensure that you have the db2hlq.SDSNL002 library installed in the CICS STEPLIB concatenation, which is required to use the DB2 JDBC driver.

- Ensure that your JDBC driver supports OSGi. You should use IBM Data Server Driver for JDBC and SQLJ Version 3.62, or Version 4.12 in your system. Reference the CICS Transaction Server for z/OS 4.2 detailed system requirements website for information about how to install the required authorized program analysis reports (APARs):


JVM server setup

To provide access to the DB2 OSGi bundle versions of the JDBC and SQLJ drivers, create a new JVM profile for your JVM server:

- Add the location of the lib directory for the appropriate DB2 driver to the LIBPATH_SUFFIX option.

- Add either the DB2 JDBC 3.0 or the JDBC 4.0 middleware bundle, together with the DB2 license bundle, to the OSGI_BUNDLES option. You can specify only one version of the JDBC bundle in the framework.

If you are updating an existing JVM server, disable and enable the JVMSERVER resource to restart the JVM and apply the changes.
Example 9-16 shows these settings in our example JVM profile that we used in the following JDBC and SQLJ deployment scenarios.

Example 9-16  An example JVM profile

JAVA_HOME=/usr/lpp/java/J6.0.1_64
LIBPATH_SUFFIX=/pp/db2v9/D081006/db2910_jdbc/lib
OSGI_BUNDLES=/pp/db2v9/D081006/db2910_jdbc/classes/db2jcc4.jar,/pp/db2v9/D081006/db2910_jdbc/classes/db2jcc_license_cisuz.jar

9.4.2 Eclipse integrated development environment setup

To develop and deploy the Java applications, we will be working in our Eclipse integrated development environment (IDE) with the CICS Explorer SDK installed. However, first of all it is necessary to transfer the JDBC OSGi bundle JAR file to the workstation and add this to the Eclipse Target Platform so that the JDBC interfaces are made available:

1. Using FTP in binary mode, transfer the db2jcc4.jar from zFS to a directory on your workstation.
2. Then, in Eclipse, click Window → Preferences → Target Platform:
   a. Select the CICS TS V4.2 Runtime Target definition and click Edit.
   b. In the Locations tab, click Add and select the folder containing the db2jcc4.jar file.
   c. In the Content tab, select the com.ibm.db2.jcc plug-in and then click Finish and OK. See Figure 9-2 on page 303.
In this section, we describe how to create a JDBC application and deploy it as an OSGi bundle application in CICS. The sample JDBC program from 9.2.1, "Example of a simple JDBC application" on page 281 is used as our example.

1. Create the plug-in project as follows:
   a. Create a new plug-in project in Eclipse called `JDBCSample` and copy or transfer the `JDBCSample.java` code (see Example 9-2 on page 281) into the package, `com.ibm.cicsjava`. 

---

**Note:** For more information about how to set up and use a target platform with the CICS Explorer SDK, see 6.1.1, “Set the target platform” on page 162.

---

**9.4.3 Deploying our JDBC sample application**

In this section, we describe how to create a JDBC application and deploy it as an OSGi bundle application in CICS. The sample JDBC program from 9.2.1, "Example of a simple JDBC application" on page 281 is used as our example.

1. Create the plug-in project as follows:
   a. Create a new plug-in project in Eclipse called `JDBCSample` and copy or transfer the `JDBCSample.java` code (see Example 9-2 on page 281) into the package, `com.ibm.cicsjava`. 

---

**Figure 9-2   Adding JDBC plug-in to the target environment**

---
Note: For further details about creating plug-in projects in Eclipse, refer to 6.1.2, “Create a plug-in project” on page 165.

b. Modify the plug-in dependencies as follows:
   i. Open the MANIFEST.MF file in the META-INF folder.
   ii. Add the com.ibm.cics.server package to provide access to the JCICS API, following the advice given in 6.1.3, “Edit the plug-in manifest file” on page 168.
   iii. Add the com.ibm.db2.jcc package to provide access to the JDBC API.
   iv. The imports in the manifest should now look like Example 9-17.

   Example 9-17  Imported packages

   
   Import-Package: com.ibm.cics.server;version="[1.300.0,2.0.0)", com.ibm.db2.jcc

   v. Add a new line in MANIFEST.MF.

   CICS-MainClass: com.ibm.cicsjava.JDBCSample

   This statement is required to enable JDBCSample to be registered as an OSGi service in CICS during JVM server initialization so that it can be started as a CICS Java program.

   Finally, the MANIFEST.MF should look like Example 9-18.

   Example 9-18  The MANIFEST.MF file

   
   Manifest-Version: 1.0
   Bundle-ManifestVersion: 2
   Bundle-Name: JDBCSample
   Bundle-SymbolicName: JDBCSample
   Bundle-Version: 1.0.0
   Bundle-RequiredExecutionEnvironment: JavaSE-1.6
   Import-Package: com.ibm.cics.server;version="[1.300.0,2.0.0)", com.ibm.db2.jcc
   CICS-MainClass: com.ibm.cicsjava.JDBCSample

2. Now, create a CICS bundle project as follows in your Eclipse IDE:
   a. Create the project

   New → Project → CICS resource → CICS Bundle project assign a project name: CICSJDBCBundle
b. Include the OSGi project in the CICS bundle:
   i. Right click the JDBCSample plug-in project and select New → other → CICS Resources → Include OSGi project in bundle to add the OSGi plug-in project to the CICS bundle project.
   ii. In the next window, click the plug-in project name, then input the JVM server name: JVMSJDBC. The wizard generates a bundle file name JDBCBundle osgibundle.

3. Deploy the CICS bundle to z/OS UNIX file system:
   a. Right click the bundle project and select → Export to z/OS UNIX File System

   When the deployment finishes, these files are found on z/OS:
   - /u/cicsrs4/cicbundle/CICSJDBCBundle/JDBCSample osgibundle
   - /u/cicsrs4/cicbundle/CICSJDBCBundle/META-INF/cics.xml
   - /u/cicsrs4/cicbundle/CICSJDBCBundle/JDBCSample 1.0.0.jar

4. Create the CICS resource to run the sample

   To run the sample in CICS, you need the following resources:
   - JVMSERVER
   - BUNDLE
   - PROGRAM
   - TRANSACTION

   To create the JVMSERVER, we first created a JVM profile in the zFS file, /u/cics/JVMProfiles/JVMJDBC, as detailed in 9.4.1, “CICS region setup” on page 301.
We then created a CICS JVM server definition, JVMSJDBC, as shown in Figure 9-3.

```
DEFINE JVMSERVER(JVMSJDBC) GROUP(TSTGRP)
OVERTYPE TO MODIFY CICS RELEASE = 0670
CEDA Define JVmserver(JVMSJDBC)
  JVmserver ==> JVMSJDBC
  Group ==> TSTGRP
  DEScRIPTION ==> JVM server for JDBC bundles
  Status ==> Enabled
 Jvmprofile ==> JVMJDBC
           (Mixed Case)
  Lerunopts ==> DFHAXRO
  Threadlimit ==> 015

......
SYSID=REDB APPLID=EPREDB
DEFINE SUCCESSFUL TIME: 13.03.44 DATE: 03/01/13
PF 1 HELP 2 COM 3 END  6 CRSR 7 SBH 8 SFH 9 MSG 10 SB 11 SF 12 CNCL
```

Figure 9-3   CICS JVM server definition

We then created a CICS bundle definition, JDBCBNDL, as shown in Figure 9-4.

```
DEFINE BUNDLE(JDBCBNDL) GROUP(TSTGRP)
OVERTYPE TO MODIFY CICS RELEASE = 0670
CEDA Define Bundle( JDBCBNDL )
  Bundle ==> JDBCBNDL
  Group ==> TSTGRP
  DEScrition ==>
  Status ==> Enabled
  Bundledir ==> /u/cicsrs4/cicsbundle/CICSJDBCBundle
           (Mixed Case)

......
SYSID=REDB APPLID=EPREDB
DEFINE SUCCESSFUL TIME: 13.03.44 DATE: 03/01/13
PF 1 HELP 2 COM 3 END  6 CRSR 7 SBH 8 SFH 9 MSG 10 SB 11 SF 12 CNCL
```

Figure 9-4   CICS bundle definition
Next, we created the transaction definition, JDBC, as shown in Figure 9-5.

![Figure 9-5  CICS transaction SQLJ]

Lastly, we created a program definition, as shown in Figure 9-6.

![Figure 9-6  CICS program SQLJSAMP]
Now, install the resources by using the following command:

CEDA INSTALL GROUP(TSTGRP)

To run the sample, enter the JDBC transaction ID on a 3270 terminal and you should get the following output (Figure 9-7).

```
JDBC
**** JDBCsample begin *****
   Loaded the JDBC driver
   getConnectionjdbc:default:connection Finished
SELECT EMPNO, FIRSTNME FROM DSN8910.EMP WHERE WORKDEPT='A00'
Employee number = 000010    FirstName = CHRISTINE
Employee number = 000110    FirstName = VINCENZO
Employee number = 000120    FirstName = SEAN
**** CICS JAVA JDBC EXAMPLE FINISHED *****
```

Figure 9-7  JDBC transaction output

Whenever you change your bundle application, reinstall the CICS bundle to enable your changes.

### 9.4.4 Deploying our SQLJ sample application

In this part, you see how to build an SQLJ application and deploy it as an OSGi bundle application in CICS. The program in 9.3.1, “Example of a simple SQLJ application” on page 292 is used as our example.

1. To begin, set up the environment to run SQLJ

   SQLJ is a utility on z/OS to translate SQLJ statements in source program. To run this, the db2jcc.jar and sqlj4.zip files are required along with the db2jcc javax.jar and db2 license files. Set this in your .profile on your default path in UNIX System Services, as shown in Example 9-19.

   ```
   $STEPLIB
   export LIBPATH=/pp/db2v9/D081006/db2910_jdbc/lib:$LIBPATH
   export PATH=$PATH:/usr/lpp/java/J6.0.1_64/bin:/pp/db2v9/D081006/
   db2910_jdbc/bin
   export CLASSPATH=/pp/db2v9/D081006/db2910_jdbc/classes/
   db2jcc4.jar:/pp/db2v9/D081006/db2910_jdbc/classes/db2jcc javax.jar:
   ```
2. Translate the SQLJ and bind the package:
   a. Create a folder on UNIX System Services and by using FTP in ASCII
      mode, put the SQLJSample.sqlj (see Example 9-7 on page 292) into that
      folder.
   b. Translate the SQLJ source by using the following UNIX System Services
      command:

      sqlj -compile=false SQLJSample.sqlj

      This translates SQLJ clauses in the program in to pure Java statements. It
      generates a .java source file and a serialized profile:

      - SQLJSample.java
      - SQLJSample_SJProfile0.ser
   c. Customize the serialized profile and bind the package by running the
      `db2sqljcustomize` command in UNIX System Services:

      db2sqljcustomize -user CICSRS4 -password PWDRS4 -url
      jdbc:db2://wtsc66.itso.ibm.com:37890/DB9E -bindoptions
      "ACTION(REPLACE)" SQLJSample_SJProfile0.ser

      You see the result that is shown in Example 9-20.

      **Example 9-20   Command result**

      | Command        | Output                                    |
      |----------------|-------------------------------------------|
      | jcc sqlj       | Begin Customization                      |
      | jcc sqlj       | Loading profile: SQLJSample_SJProfile0    |
      | jcc sqlj       | Customization complete for profile        |
      | jcc sqlj       | Begin Bind                                |
      | jcc sqlj       | Loading profile: SQLJSample_SJProfile0    |
      | jcc sqlj       | User bind options: ACTION(REPLACE)       |
      | jcc sqlj       | Driver defaults(user may override):       |
      |                | BLOCKING ALL                              |
      |                | VALIDATE BIND                             |
      | jcc sqlj       | Fixed driver options: DATETIME ISO        |
      |                | DYNAMICRULES BIND                         |
      | jcc sqlj       | Binding package SQLJSA01 at isolation level |
      |                | UR                                       |
      | jcc sqlj       | Binding package SQLJSA02 at isolation level |
      |                | CS                                       |
      | jcc sqlj       | Binding package SQLJSA03 at isolation level |
      |                | RS                                       |
      | jcc sqlj       | Binding package SQLJSA04 at isolation level |
      |                | RR                                       |
      | jcc sqlj       | Bind complete for SQLJSample_SJProfile0   |
This command binds four packages in plan DSNJCC:

- DB9E..SQLJSA01 for isolation level UR
- DB9E..SQLJSA02 for isolation level CS
- DB9E..SQLJSA03 for isolation level RS
- DB9E..SQLJSA04 for isolation level RR

3. A JDBC program might select which package to use by using the 
   `Connection.setTransactionIsolation()` method; the default is CS. Perform 
   the following instructions to create the plug-in project:

   a. Create a new plug-in project in Eclipse and copy or transfer 
      `SQLSample.java` code into package `com.ibm.cicsjava` in the `src` folder of 
      this project. For more information about creating plug-in projects in 
      Eclipse, see 6.1.2, “Create a plug-in project” on page 165.

   b. Transfer the serialized profile `SQLJSample_SJProfile0.ser` from 
      UNIX System Services to your workstation, and copy into the `src` folder of 
      the new plug-in project.

   **Note:** If you are using FTP to transfer the serialized profile, ensure that 
   binary mode is used.

   c. Modify the plug-in dependencies as follows:

   i. Open the `MANIFEST.MF` file in the `META-INF` folder.

   ii. Add the `com.ibm.cics.server` package to provide access to the JCICS 
       API, following the advice given in 6.1.3, “Edit the plug-in manifest file” 
       on page 168.

   iii. Add the following SQLJ and JDBC packages:

   - `com.ibm.db2.jcc.sqlj`
   - `com.ibm.db2.jcc`
   - `sqlj.runtime`
   - `sqlj.runtime.error`
   - `sqlj.runtime.profile`
   - `sqlj.runtime.profile.ref`
   - `sqlj.runtime.ref`

   iv. The imports in the manifest should now look like Example 9-21.

   **Example 9-21   Imported packages**

   Import-Package: com.ibm.cics.server;version="[1.300.0,2.0.0)",
   com.ibm.db2.jcc.sqlj,
   com.ibm.db2.jcc,
   sqlj.runtime,
   sqlj.runtime.profile,
v. Add the CICS Main class definition to the MANIFEST.MF as follows:

   CICS-MainClass: com.ibm.cicsjava.SQLJSample

This statement enables the com.ibm.cicsjava.SQLJSample to be automatically registered as an OSGi service in CICS during its installation, allowing it to be started as a CICS Program.

Finally, the MANIFEST.MF looks like Example 9-22.

Example 9-22  Result of the MANIFEST.MF

<table>
<thead>
<tr>
<th>Manifest-Version: 1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bundle-ManifestVersion: 2</td>
</tr>
<tr>
<td>Bundle-Name: SQLJSample</td>
</tr>
<tr>
<td>Bundle-SymbolicName: SQLJSample</td>
</tr>
<tr>
<td>Bundle-Version: 1.0.0</td>
</tr>
<tr>
<td>Bundle-RequiredExecutionEnvironment: JavaSE-1.6</td>
</tr>
<tr>
<td>Import-Package: com.ibm.cics.server;version=&quot;[1.300.0,2.0.0)&quot;, com.ibm.db2.jcc.sqlj, com.ibm.db2.jcc, sqlj.runtime, sqlj.runtime.profile, sqlj.runtime.profile.ref, sqlj.runtime.ref, sqlj.runtime.error</td>
</tr>
<tr>
<td>CICS-MainClass: com.ibm.cicsjava.SQLJSample</td>
</tr>
</tbody>
</table>

4. Include OSGi project in CICS bundle:
   a. Include OSGi project in the previous created CICS bundle project, CICSJDBCBundle.
      i. Right click the CICSJDBCBundle project and select New → other → CICS Resources → Include OSGi Project in Bundle.
      ii. Select the plug-in project, SQLJSample, then input the JVM server name, JVMSJDBC, that is used in our CICS region.
A new OSGi bundle file, SQLJBundle osgibundle, is now generated in the CICSJDBCBundle project.

b. Deploy the CICS bundle to the z/OS UNIX file system:
   i. Right click the CICS bundle project → Export to z/OS UNIX File System.

5. Create the CICS resource to run the sample

You are now ready to run the sample in CICS. To do this, we define the following new resources:

– Transaction SQLJ. See Figure 9-8
– Program SQLJSAMP. See Figure 9-9 on page 313.

```
DEF TRAN(SQLJ) GROUP(TSTGRP)
OVERTYPE TO MODIFY
CEDA DEFINE TRANSACTION( SQLJ )
  TRANSACTION ==> SQLJ
  GROUP      ==> TSTGRP
  DESCRIPTION ==> sql test
  PROGRAM    ==> SQLJSAMP
  TWSIZE     ==> 00000              0-32767
  PROFILE    ==> DFHCICST
  PARTITIONSET ==>
  STATUS     ==> Enabled            Enabled | Disabled
  ....
  SYSID=REDB APPLID=EPRDB
DEFINE SUCCESSFUL TIME: 13.04.42 DATE: 03/01/13
PF 1 HELP 2 COM 3 END 6 CRSR 7 SBH 8 SFH 9 MSG 10 SB 11 SF 12 CNCL
```

*Figure 9-8  CICS transaction SQLJ*
Figure 9-9 shows the CICS program SQLJSAMP.

We also reused the existing JVM server resource, JVMSJDBC, which was defined in 9.4.2, “Eclipse integrated development environment setup” on page 302.

Having defined these resources, the transaction and program definitions need to be installed and the bundle must be refreshed. Issue the following commands from a 3270 terminal:

a. **CEDA INSTALL PROG(SQLJSAMP) GROUP(TSTGRP)**
b. **CEDA INSTALL TRAN(SQLJ) GROUP(TSTGRP)**
c. **CEMT SET BUNDLE(JDBCBNDL) DISABLE**
d. **CEMT DISCARD BUNDLE(JDBCBNDL)**
e. **CEDA INSTALL BUNDLE(JDBCBNDL)**
Now you are ready to run the sample by using transaction SQLJ from a 3270 terminal. The output should look like what is shown in Figure 9-10.

```
SQLJ
****  SQLJ example begin  *****
  successful driver load, version 4.13.110
  successful connect; autocommit is false
SELECT EMPNO,FIRSTNME FROM DSN8910.EMP WHERE WORKDEPT=A00
  successful execution of SELECT
    EmployNumber: <000010> FirstName: <CHRISTINE>
    EmployNumber: <000110> FirstName: <VINCENZO>
    EmployNumber: <000120> FirstName: <SEAN>
  successful processing of Iterator
  successful close of iterator
****  SQLJ example complete  *****
```

*Figure 9-10  SQLJ transaction output*

**Updating an SQLJ application**

Each time that you modify your SQLJ application, you need to do the following steps:

1. On z/OS, run the SQLJ command to translate SQLJ classes.
2. Run the db2sqljcustomize command to customize the serialized profile and bind the package.
3. In Eclipse, refresh source code from the generated .java program in step 1 and refresh the serialized profile from the .ser generated in step 2.
4. In Eclipse, redeploy the CICS bundle to z/OS.
5. In CICS, discard, reinstall, and enable the BUNDLE resource to refresh the OSGi bundle.
Migration

This chapter describes the migration and conversion process to enable your pre CICS Transaction Server (TS) V4.2 Java applications to run in either a Java virtual machine (JVM) pool or a JVM server environment. We introduce the concepts and then demonstrate examples of each step of the migration by using both JVM pool and JVM server technology.

In our scenario, we have a Java application running in a CICS TS V4.1 region and want to migrate it to a CICS TS V4.2 region using JVM pool technology. When it is up and running successfully, the application is converted to run under a JVM server. There are three different migration options available to convert to a JVM server. Your specific circumstances and application determine which is best for you. We demonstrate all three and offer guidance on when to use each technique.

You might also migrate directly from CICS TS V4.1 to a JVM server in CICS TS V4.2. However, this chapter concentrates on migrating first to the JVM pool environment.

To migrate an application to CICS TS V4.2, we need a Java application that is running in a pre-v4.2 release of CICS. The previous Java IBM Redbooks publication, *Java Application Development for CICS*, SG24-5275, has what is needed: several Java applications written for a pre-v4.2 release of CICS that runs in a JVM pool environment.
10.1 Existing CICS TS V4.1 test application

For our scenario, we use the TRADER2 sample Java application running in a CICS TS V4.1 region and migrate it to a CICS TS V4.2 region. The TRADER2 sample application was downloaded from the previous CICS Java Redbooks publication, *Java Application Development for CICS*, SG24-5275.

Chapter 7 of *Java Application Development for CICS* describes the process of taking a 3270-based COBOL application *Trader*, and converts the COBOL programs to Java programs (using the Java class library for CICS (JCICS) application programming interface (API)). The Redbooks publication uses a phased approach to convert the front end presentation logic from a 3270-based COBOL program to a web browser Java based program. The Trader-1 sample builds the Presentation Logic front end in Java using the JCICS API to call the original COBOL-based business logic via a Commarea. Next, the TRADER2 sample creates a Java program to replace the COBOL business logic paring it with the Java front end from the Trader-1 sample. Finally, the Trader-3 sample converts the Virtual Storage Access Method (VSAM) files to DB2 using Java Database Connectivity (JDBC) calls.

For this demonstration, we use the TRADER2 application, which consists of the Java presentation logic program, *TRADERPJ*, which calls the Java business logic program, *TRADERBL*. The original Commarea is used as the interface. Complete details about the TRADER2 application and the conversion process can be found in Chapter 7 of *Java Application Development for CICS*.

You can download and test any of the three Trader applications described in Chapter 7 of *Java Application Development for CICS* from the following link:

ftp://www.redbooks.ibm.com/redbooks/SG245275/
The original 3270 application is described in *Java Application Development for CICS*. Its components are shown in Figure 10-1.

![Figure 10-1 Original Trader COBOL application](image1)

Figure 10-2 shows the new TRADER2 Java application components that replace the original COBOL application.

![Figure 10-2 TRADER2 application converted to Java](image2)

Both applications are installed in our regions using a shared CICS system definition data set (CSD) and reference the same VSAM Files. In addition to the original CSD definitions for the 3270-based COBOL application, a TCPIPSERVICE port was added to the CICS TS V4.1 and CICS TS V4.2 regions to allow the new browser front end to connect to CICS. Figure 10-4 on page 321 shows the new Java program definitions that are needed to define the Java programs that are shown in Figure 10-2.
10.2 Migration process overview

Your goal should be to convert from JVM pools to JVM server as soon as possible. However, our aim in this book is to provide a more gradual process, migrating JVM pooled applications to CICS TS V4.2 before converting them to JVM server.

It is important to highlight that IBM issued a statement of direction in the CICS TS V4.2 announcement letter. The letter states that you have to migrate to the JVM server environment. See CICS TS V4.2 Announcement Letter 211-080.

Statement of direction: IBM intends a future release of CICS TS to discontinue support for both: (i) session beans using Enterprise JavaBeans (EJB); and (ii) the Java pool infrastructure. Customers are encouraged to migrate Java applications to the new JVM server infrastructure.

During the review cycle of this book, CICS TS V5.1 was announced and we can confirm that JVM pools and EJBs were removed in CICS TS V5.1.

One consideration when migrating applications to a JVM server is that unlike JVM pools, a JVMSERVER is a long-lived CICS resource and can be considered a CICS System component, such as DB2 Connections, WebSphere MQ Trigger Monitors, and TCPIPService ports. After you set up your region with a JVMSERVER resource, you can choose to run all your Java applications for that region in that same JVMSERVER. Or you can choose to host different applications in different JVM servers, perhaps to provide better scalability or isolation of those particular applications.

Both the JVM pool and JVM server can coexist in CICS TS V4.2.

10.3 Migration to CICS TS V4.2 using a JVM pool

For this scenario, we run a CICS TS V4.1 and a CICS TS V4.2 region side by side allowing us to make comparisons. We cloned the CICS TS V4.1 region, and then upgraded it to CICS TS V4.2. The CICS TS V4.2 upgrade process is out of the scope of this chapter. We are only concerned with the changes required to the Java application and its components.

Both regions share a CSD. The original 3270 COBOL version of Trader is installed in the CICS TS V4.1 region along with the TRADER2 components shown in Figure 10-1 on page 317 and Figure 10-2 on page 317. All information
about running TRADER2 in CICS TS V4.1 came from *Java Application Development for CICS*, SG24-5275, and is not described in this book.

When TRADER2 was running in CICS TS V4.1, it was converted to run in CICS TS V4.2 using the steps outlined in the CICS TS V4.2 Information Center. See “Upgrading the Java Environment”. We chose not to duplicate those steps here because the information center is always the most up-to-date reference.

Below is a summary of the steps performed and the actions taken to upgrade the TRADER2 application from a CICS TS V4.1 region to a CICS TS V4.2 region. Steps 1 through 11 are described in section, “Upgrading from IBM 31-bit SDK for z/OS, Java Technology Edition, Version 6”:

1. No changes required for TRADER2
2. No changes required for TRADER2
3. Already installed
4. Created a new profile directory for the CICS TS V4.2 region: DFHJVMPPR
   a. Updated JAVA_HOME with /usr/lpp/java/J6.0.1_64/
   b. Deleted CICS_HOME
   c. Reviewed all parameters (parms) against “JVM profiles: options and samples”
5. Access verified and OK
6. Updated SIT Parm USSHOME. See Figure 10-6 on page 322
7. Updated SIT Parm JAVAPROFILEDIR. See Figure 10-6 on page 322
8. Our regions are using REGION=0M and thus have NOLIMIT
   See “Estimating, checking, and setting MEMLIMIT” for tuning
9. Runtime testing comparing both the CICS TS V4.1 and CICS TS V4.2 regions during execution:
   a. Confirmed. See Figure 10-7 on page 323 and Figure 10-8 on page 323 for CICS Explorer snapshots of the JVM pools and JVMs running in them
   b. Confirmed. See Figure 10-9 on page 323 for a CICS Explorer snapshot of the JVM class caches
   c. Confirmed. See Figure 10-10 on page 324 through Figure 10-12 on page 326 for some execution runs
   d. Confirmed. No changes for JVM pools. See Figure 10-6 on page 322 for the current JVMCCSIZE and JVMCSTART settings
10. No problems found with TRADER2 running under CICS TS V4.2. However, if you are having problems, review the items in this section.

11. Not applicable for this scenario because we only needed one region.

We base our upgraded JVM profile on the sample profile for applications, DFHJVMPR. DFHJVMCD is used internally by CICS Java programs. Do not use either of these JVM profiles for a JVM server. If you are interested in seeing where DFHJVMCD can be used, you can use the CICS Explorer to filter all programs that use JVM profile DFHJVMCD, as shown in Figure 10-3.

![Figure 10-3  Programs that use JVM profile DFHJVMCD](image-url)
Figure 10-4 shows the Java tab for the TRADER2 program definitions. The CSD is shared between the CICS TS V4.1 and CICS TS V4.2 regions, thus the same program definitions are used and required no changes for the upgrade. Both applications use a JVM profile of the same name, DFHJVMPR, but its contents might change between releases. Therefore, have a separate version of DFHJVMPR for each version of CICS.

<table>
<thead>
<tr>
<th>Java Virtual Machine (JVM)</th>
<th>Java Virtual Machine (JVM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operate program under control of a JVM</td>
<td>Operate program under control of a JVM</td>
</tr>
<tr>
<td>Fully qualified main Java class name to be run</td>
<td>Fully qualified main Java class name to be run</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Environment</th>
<th>Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Java Program can run in a JVM Server, or in a JVM Pool with options specified in a JVM Profile.</td>
<td>The Java Program can run in a JVM Server, or in a JVM Pool with options specified in a JVM Profile.</td>
</tr>
<tr>
<td>Use a JVM Server:</td>
<td>Use a JVM Server:</td>
</tr>
<tr>
<td>Use the default JVM Profile (DFHJVMPR)</td>
<td>Use the default JVM Profile (DFHJVMPR)</td>
</tr>
<tr>
<td>Use a named JVM Profile: DFHJVMPR</td>
<td>Use a named JVM Profile: DFHJVMPR</td>
</tr>
</tbody>
</table>

*Figure 10-4   TRADERPJ and TRADERBL program definitions via shared CSD*
Figure 10-5 shows the two different versions of JVM profile, DFHJVMPR, used by the TRADERPL and TRADERBL programs in a side-by-side comparison. The view tabs show their actual storage locations, which highlight that the CICS TS V4.1 profile was copied and updated, thus allowing the program definitions to remain unchanged.

Figure 10-5   Comparison of DFHJVMPR: CICS TS V4.1 on left and CICS TS V4.2 on right

In Figure 10-6, the CICS Explorer **Show SIT Parameters → COMBINED** option was used to show the original SIT table definitions with their applied parameter overrides. As you can see from the comparison display, the **JVMCCSIZE** and **JVMCCSTART** parameters that are associated with the JVM pool class cache were left unchanged because we were not adding or deleting any Java applications or workload during the upgrade. The **USSHOMEDIR** and **JVMPROFILEDIR** parameters were the key parameters changed for the upgrade.

Figure 10-6   Comparison of SIT Parameters: CICS TS V4.1 on the left and CICS TS V4.2 on the right
After the conversion, the sample Java based transactions were run on both the CICS TS V4.1 and CICS TS V4.2 regions. Figure 10-7 is the CICS Explorer JVM pools view showing the JVM pools being used by both the original CICS TS V4.1 region and the converted CICS TS V4.2 region. Column Sjgpeakjvms shows the peak number of JVMs in the JVM pool.

Figure 10-7  CICS Explorer view of JVM pools in CICS TS V4.1 and CICS TS V4.2

Figure 10-8 gives more details about each of the JVMs that were identified in Figure 10-7. As you can see, there are three JVMs in the CICS TS V4.1 region and two JVMs in the CICS TS V4.2 region. This can be useful in mapping your JVM activity to the JVM profiles that define their characteristics. Column Allocage is zero, which means that no currently running tasks are using the JVMs. Age shows how old each JVM is.

Figure 10-8  CICS Explorer view of JVMs running in CICS TS V4.1 and CICS TS V4.2

Figure 10-9 shows the status of the class cache. Both CICS TS V4.1 and CICS TS V4.2 use profile DFHJVMCD to initialize and terminate the shared class cache for pooled JVMs, hence the similar results regarding the class cache status.

Perform before and after testing to verify that you have enough class cache storage available. You can use the JVMCCIZE SIT parm to set the initial value and you can use the CEMT PERFORM CLASSCACHE RELOAD with the CACHESIZE option to dynamically create a new shared class cache.

For more information, see “Adjusting the size of the shared class cache” in the CICS Information Center.

Figure 10-9  CICS Explorer view of JVM class caches in CICS TS V4.1 and CICS TS V4.2
In Figure 10-10 through Figure 10-12 on page 326, there are several screen captures showing the TRADER2 application running in both CICS TS V4.1 and CICS TS V4.2 simultaneously. Both regions share application files, thus Figure 10-11 on page 325 shows buying five shares, changing the shares owned from 46 to 51. Figure 10-12 on page 326 shows selling four shares, changing the shares owned from 51 to 47. TCPIPS Port 18141 is the CICS TS V4.1 region and Port 18142 is the CICS TS V4.2 region.

Figure 10-10 shows the initial TRADER2 Company Selection panel running in CICS TS V4.2.

![Company Selection](image)

*Figure 10-10  TRADER2 initial menu running in CICS TS V4.2*
Figure 10-11 shows a purchase of five shares raising the portfolio up to 51 shares owned.

Figure 10-11  TRADER2 buying five shares running in CICS TS V4.2
Switching over to the CICS TS V4.1 region, using the same shared customer application files, a sell request is made for four shares, changing the portfolio shares owned down to 47 shares. See Figure 10-12.

![Sell Shares](image)

This demonstration of both CICS regions operating in tandem completes the migration from CICS TS V4.1 to CICS TS V4.2. As you can see from the results, not only is it easy to migrate, but both can coexist running the same code and sharing resources.

You might feel the need to make further adjustments to your JVM pooled environment. You can find several topics on managing the JVM pool environment along with how to manage the class cache in the CICS TS Information Center. See the section that is titled: Managing pooled JVMs.

Next, we convert the TRADER2 application to run in a JVM server.
10.4 Converting from a JVM pool to a JVM server

This section converts the TRADER2 application running in a CICS TS V4.2 region using JVM pools to the new JVM server runtime environment. You could have skipped the JVM pool steps altogether and migrated directly to a JVM server by following the steps in the CICS TS V4.2 Information Center under the topic: “Upgrading the Java Environment”.

For this section, the same procedure that was performed in 10.3, “Migration to CICS TS V4.2 using a JVM pool” on page 318 is followed. However, instead of using JVM profile DFHJVMPR, we use DFHOSGI.

The conversion process comes from two sources: The CICS TS V4.2 Information Center, and the CICS Explorer SDK help section “CICS Java Developer Guide”. The CICS Information Center gives a high-level overview explaining the three methods of “Migrating applications using the CICS Explorer SDK”. The CICS Explorer SDK help section also explains the same three methods, using different names to describe them, but it goes into more detail. You must use the CICS Explorer SDK to package your Java applications as OSGi bundles. See topic “Packaging existing applications to run in a JVM server” in your CICS Explorer SDK help section in the “CICS Java Developer Guide”.

You can repackage your Java applications as OSGi bundles for execution in a JVM server in the following ways:

- Conversion: Converting an existing Java project to a plug-in project
- Injection: Importing the contents of a Java ARchive (JAR) file into an OSGi plug-in project
- Wrapping: Importing a binary JAR file into an OSGi plug-in project

The three methods are described as follows:

1. Injection: This creates one large bundle and exports all packages. It has the advantage that the migrated JAR file can be deployed as either an OSGi bundle into a JVM server or into a standard class path environment such as a CICS JVM pool.

2. Wrapping: This does not modify the original binary JAR file. Therefore, it might offer the advantage of requiring less testing, or keeping compliance where licenses do not allow modification. However, the wrapped JAR file will not run in a standard class path environment such as a CICS JVM pool.

3. Conversion: This takes an existing Java program and converts it to an OSGi bundle (plug-in project). This is the preferred option for migrating to the JVM server and offers the most flexibility for future development.
Note: CICS uses Open Transaction Environment (OTE) to run up to 256 parallel tasks in a single JVM server using T8 task control blocks (TCBs).

Threadsafe considerations apply when running in the JVM server environment because the CICS tasks are running in parallel as threads in the same JVM server process. Not only is the JVM shared by all CICS tasks, which might be running multiple applications concurrently, all static data and static classes are also shared.

Therefore, all Java applications must be threadsafe when running in a JVM server.

10.4.1 Before you begin: Setting up the target environment

Before you start, ensure that your target platform is set to the release of CICS that you will be running.
Figure 10-13 shows selecting **Window → Preferences**. Then, from the Preferences window, select **Plug-in Development → Target Platform**, and click **Add**. Then, from the **New Target Definition** window, select **Template**. Then, choose the release of CICS that you will export to. Click **Next**, then **Finish**.

*Figure 10-13  Set the target platform: CICS release*
On the Preferences window, ensure that you check the target releases of CICS, as shown in Figure 10-14.

![Preferences window](image)

Figure 10-14 Choosing the target CICS release

You are now ready to start the migration process from a JVM pool to a JVM server.
10.4.2 Migration to a JVM server: Conversion method

The conversion method assumes that you already have your Java classes in Eclipse as a Java project. If not, you might want to create a Java project for your source-code, but this is outside of the scope of this book.

An Eclipse IDE with the CICS Explorer SDK is capable of developing Java applications to run in any supported release of CICS. Therefore, if you already have a Java application in the SDK for a release of CICS before CICS TS V4.2, you can use the conversion method to build and package a CICS bundle project and export it to a CICS TS V4.2 region to run in a JVM server environment.

The following steps, which are taken from the CICS Explorer SDK help section under the “CICS Java Developer Guide” in section “Converting an existing Java project to a plug-in project”, show how to convert the TRADER2 application from a JVM pool to a JVM server execution environment that runs in a CICS TS V4.2 region.

The TRADER2 application is already in our CICS Explorer SDK and installed and running in both a CICS TS V4.1 region and a CICS TS V4.2 region (as JVM pooled applications). Therefore, the conversion method is a good choice to package and deploy the application to a JVM server in CICS TS V4.2.
To start the conversion process, go to the CICS Explorer SDK where the TraderTwo application is available in the Project Explorer view (shown in Figure 10-15).

The CICS Explorer SDK provides the functionality to convert the project for you. Right-click the TraderTwo project in the Package Explorer and select **Configure → Convert to Plug-in Projects**, as shown in Figure 10-15.

*Figure 10-15  Convert an existing Java project to an OSGi project*
Select your Java project and click **Finish** to complete the conversion process, as shown in Figure 10-16.

![Convert project selection panel](image)

**Figure 10-16  Convert project selection panel**

Your existing Java project has been converted to an OSGi plug-in project. From here, we can create a CICS bundle project that references your new Java OSGi bundle project, and export it to CICS.

Before that however, there are some customizations necessary to prepare the application for the CICS OSGi run time.
In the Package Explorer, expand your CICS Java plug-in project and double-click the MANIFEST.MF file to open the Plug-in Manifest Editor. Ensure that you are viewing the Overview tab on the bottom and then delete the .qualifier portion of the Version field, as shown in blue in Figure 10-17.

Figure 10-17  Remove the .qualifier portion
Still on the Overview tab, click **Add** in the Execution Environments section and select the minimum level of Java required for this application. In our case, we are assuming that our application uses some Java 6 API, and therefore we select **JavaSE-1.6**, as shown in Figure 10-18.

![Figure 10-18 Set the execution environment](image)

The unique steps for the conversion method are done. The remaining conversion steps have been combined with the other two conversion processes and are covered in 10.4.5, “Common steps for all three JVM server migration methods” on page 342.

Before continuing with the common section, review the following regarding the conversion method migration.
Because all three migration processes are similar and have an overlap of common steps, the rest of the migration is in 10.4.5, “Common steps for all three JVM server migration methods” on page 342.

10.4.3 Migration to a JVM server: Injection method

The *injection method* assumes that you already have your Java application in a JAR file available on your workstation that you can import into the CICS Explorer SDK. This section demonstrates the process of packaging the JAR application into a CICS bundle for export to a CICS TS V4.2 region and execution in a JVM server.

The following steps, which are taken from CICS Explorer SDK help: “CICS Java Developer Guide” in section “Importing the contents of a JAR file into an OSGi plug-in project”, show how to import the TRADER2 JAR file into an OSGi bundle plug-in project.

Because the TRADER2 application is already packaged as a JAR file and is on our workstation, we can conveniently demonstrate the injection method.
To start the injection process, go to the CICS Explorer SDK and select **Window → Open Perspective → Other** and select **Plug-in Development** to open the Plug-in Development perspective. From the Project Explorer, right-click anywhere and select **New → Project** to open the New Project window and then select **Plug-in from Existing JAR Archives**, as shown in Figure 10-19.

**Note:** You can start from a Java perspective and Eclipse prompts you to switch to the Plug-in Development perspective. Therefore, you do not have to start off in the Plug-in Development perspective.

![Creating an OSGi plug-in project from an existing JAR file](image)
On the New Project window that is shown in Figure 10-19 on page 337, click **Next** to open the New Plug-in from Existing JAR Archives window, which is shown in Figure 10-20. Select **Add External** to locate the JAR file on your workstation.

*Figure 10-20  Import an existing JAR file*
On the New Plug-in from Existing JAR Archives window shown in Figure 10-20 on page 338, click Next to open the Plug-in Project Properties window (as shown in Figure 10-21). Here, we set some of the project’s properties. On the left is the initial window and on the right is the project name. Set the Execution Environment to JavaSE-1.6. Again, we assumed that Java 6 is the minimum level of API that our application will run on. Set the OSGi framework to standard. This setting ensures that the plug-in remains generic and does not target any particular implementation of OSGi.

Click Finish to complete the import and open the Plug-in Manifest Editor, as shown in Figure 10-21.

Figure 10-21  Plug-in Project Properties window

Your existing Java project has now been converted to a plug-in project. The next step is to create a CICS bundle project that references your new Java OSGi bundle project, and export it to CICS.

The unique steps for the injection method are now complete and we can package it and export to z/OS for testing. Because all three migration processes are similar, the common steps overlap. For the remaining migration steps see 10.4.5, “Common steps for all three JVM server migration methods” on page 342.
10.4.4 Migration to a JVM server: Wrapper method

The wrapper method assumes that you already have your Java application in a JAR file available on your workstation that you can import in binary format into the CICS Explorer SDK.

The wrapper method is almost identical to the injection method with one important difference: During the import process the JAR file is left compressed, and treated as a binary object.

To perform the wrapper method, follow 10.4.3, “Migration to a JVM server: Injection method” on page 336. However, on the New Plug-in from Existing JAR Archives window of Figure 10-21 on page 339, ensure that you clear the Unzip the JAR archives into the project box, as shown in Figure 10-22 on page 341.
After clearing the Unzip box, click **Finish** and follow the common steps in the next section. The wrapper method is generally used where you do not have access to the original source-code, or where local procedures do not allow tampering or alteration to the binary version of the application. Wrapping can allow the application to be deployed as an OSGi bundle, without altering the original binary.
10.4.5 Common steps for all three JVM server migration methods

In this common section, we continue by setting options that are common to all three migration methods. The options are added by using the Plug-in Manifest Editor. We build the CICS bundle project, export it to CICS, and finally set up the CICS side resources.

You should have a TraderTwo project in the Package Explorer. Expand the project and double-click the MANIFEST.MF file to open the Plug-in Manifest Editor. Select the Dependencies tab and click Add to open the Package Selection window (as shown in Figure 10-23).

**Note:** There might be slight differences between the project structures but the steps are the same.

![Setting the dependencies](image.png)
Search for and select `com.ibm.com.server` and click OK. The results should look as in Figure 10-24. CICS TS V4.2 uses version 1.300.0 of the `com.ibm.cics.server` package, so be sure to select this version as a minimum.

**Tip:** To ensure seamless migration of your application to future releases of CICS, select a range of versions for any imported packages. For example, using the following syntax ensures that your application continues to work with all versions of JCICS up to a new major version (v2.0.0), which would be considered a new interface:

```text
com.ibm.cics.server;version="[1.300.0,2.0.0)"
```

You should not expect your application to be compatible with a major version change. However, minor API additions such as 1.400.0, 1.500.0, and 1.600.0 remain compatible.

---

*Figure 10-24  Setting the dependencies results*
On the Plug-in Manifest Editor, select the MANIFEST.MF tab and add a CICS-MainClass header, as shown in Figure 10-25.

![Plug-in Manifest Editor](image)

**Figure 10-25  Add the CICS-MainClass parameter**

Make one entry into the list for each resource definition online (RDO) defined program definition that is the target of a Transaction Definition (or is XCTL or LINKed too). Notice that the names are built up from the package name and the class names, which are known as the **fully qualified package names**. Save and close the Plug-in Manifest Editor to continue.
From the Project Explorer view, right-click your project name, select **New → Project** and from the New Project window, select **CICS Bundle Project**, as shown in Figure 10-26.

![Figure 10-26  Create a CICS bundle project](image-url)
Give your project an appropriate name and click **Finish**, as shown in Figure 10-27.

*Figure 10-27  Creating a CICS bundle project*
Include the TraderTwo OSGi plug-in project, as shown in Figure 10-28, by right-clicking the new CICS bundle project, TraderTwoBundle, that was just created in Figure 10-27 on page 346. Select New → Other and from the New selection window, select Include OSGi Project in Bundle and click Next.

Figure 10-28  Including the OSGi plug-in into the CICS bundle project
From the Include OSGi Project in Bundle window, select the **TraderTwo** project and enter the name of the JVM server where the application will run. Then, click **Finish**, as shown in Figure 10-29.

![Image of Include OSGi Project in Bundle window]

*Figure 10-29  Setting the target JVM server*

As described earlier, you should have already set up a JVM server in the target CICS region. For this example, the sample JVM server, `DFH$JVMS`, is used.

You can use the same JVM server definition and profile in all regions, which eliminates any required edits at installation time.
The next step is to export the CICS bundle project, TraderTwoBundle, to z/OS and then install it in the target CICS region via a CICS bundle definition. Right-click the **TraderTwoBundle** CICS bundle project in the Project Explorer and select **Export Bundle Project to z/OS UNIX File System**, as shown in Figure 10-30.

![Figure 10-30 Exporting a CICS bundle project to z/OS UNIX System Services](image)

Select **Export to a specific location in the file system** and click **Next**.
From the Export to z/OS UNIX File System window, connect to the target z/OS system. In the Parent Directory section, select the wanted UNIX System Services directory where you want to store your exported CICS bundle. Notice in the first panel in Figure 10-31, we would like to store our bundle in a subdirectory of the Trader directory. The Bundle Directory section shows us the final directory structure that is created where the CICS Explorer SDK places the TraderTwoBundle_1.0.0 bundle. Clicking **Finish** starts the FTP process.

![Figure 10-31 FTPing the CICS bundle to a target z/OS UNIX System Services Directory](image)

Next, create an RDO bundle definition in the target CICS region and install it to activate the TRADER2 application.
In the Bundle Definitions view, which is shown in Figure 10-32, right-click any existing bundle definition and select **New** to open a New Bundle Definition window. Complete the fields as appropriate using the target library from Figure 10-31 on page 350, and click **Finish**.

![Figure 10-32 Creating a CICS bundle resource definition for TraderTwoBundle](image)

The two Java programs, TRADERPJ and TRADERBL, are currently set up to run using the JVM pool. Therefore, the program definitions must be modified. Then, the bundle and programs can be installed to activate the application.

Figure 10-33 on page 352 shows both program definitions using the CICS Explorer’s Program Definition view. The JVM pooled programs are on the left and the updates to convert them to JVM servers on the right. The distinction is made on the program definition by selecting either a JVM profile or a JVM server.
Figure 10-33  Trader program definitions: JVM pool on the left and JVM server on the right
Installing the TRADER2 group (which contains the two programs listed in Figure 10-33 on page 352, and the CICS bundle definition listed in Figure 10-32 on page 351) completes the migration process.

Notice that the actual TRADER2 group is listed on the right side of Figure 10-34.
Figure 10-34 on page 353 shows the installation of the TRADER2 Group and Figure 10-35 shows the installed resources.

Figure 10-35  TRADER2 is now defined and installed using a JVM server
We do a quick test to verify our application. Recall that the last update was the CICS TS V4.1 copy of TRADER2 leaving 47 shares. Changing the URL to the CICS TS V4.2 region's port 18142, and buying 18 shares shows a new result of 65 shares in Figure 10-36.
Finally, Figure 10-37 shows program usage stats indicating that the TRADERPJ and TRADERBL programs are now using a JVM server.

Installing a general-use JVM server for Java applications in the target CICS region provides you with an engine to run your Java programs. The setup can be found in Chapter 3, “Setting up a Java virtual machine server” on page 41. Figure 10-38 shows a comparison between the JVM profile used to run the TRADER2 application in CICS TS V4.2 (DFHJVMPR), and the JVM profile (DFHOSG1) that is used by the sample JVM server (DFH$JVMS) used to run the TRADER2 app.

Figure 10-37  Program stats showing the usage of the JVM servers

Figure 10-38  JVM pool profile on the left and JVM server profile on the right
If you want to use class cache, see 10.5, “Shared class cache in a JVM server environment” on page 359 for the steps on how to turn it on for a JVM server environment.

10.4.6 Migrating CICS TS V4.2 JVM server app to CICS TS V5.1 beta

With a copy of the CICS TS V.1 open beta available at the time of writing, it is an interesting exercise to try running the TRADER2 app in a new v5.1 region. This is not a complex application and does not test all the features and functions of a more complex application, but it does demonstrate the basic principles.

For more information, review the CICS TS V5.1 announcement letter:

The instructions for this migration are from the CICS TS V5.1 Open Beta Information Center under the “Upgrading the Java environment” section of the “Upgrading from Version 4 Release 2” chapter.

► Remove the following Java related obsolete parameters from the SIT:
  – JVMCCSIZE
  – JVMCCSTART
  – JVMLEVEL0TRACE
  – JVMLEVEL1TRACE
  – JVMLEVEL2TRACE
  – JVMUSERTRACE
  – MAXJVMTCBS

► Update the following SIT Parms:
  – JVMPROFILEDIR=/user/cics/cicsdemo/JVMProfiles51/
  – USSHOME=/usr/lpp/cicsts/cicsts.v5r1/

► Update the JVM profile

The profile in use by the TRADER2 application is DFHOSGI. The only change to it was to update the version of Java: JAVA_HOME=/usr/lpp/java/J7.0_64/

Note: This is not a CICS upgrade and migration guide and is not covering all the steps needed to migrate a CICS region. This section is looking only at the Java steps needed to migrate a simple JVM server-based Java application.
For the test to succeed, it is important that you followed the tip on page 343, which encouraged you to select a version range of the imported JCICS API (com.ibm.cics.server) in the MANIFEST.MF file.

Perform a quick test of the application, and recall that the last update was with the CICS TS V4.2 copy of TRADER2 leaving 65 shares. See Figure 10-36 on page 355. Changing the URL for the CICS TS V5.1 beta region’s port 18151 and buying 24 shares shows a new result of 89 shares (shown in Figure 10-39).
10.5 Shared class cache in a JVM server environment

In a CICS TS V4.2 region, the support for caching Java classes is now managed by Java 6 outside of the control of CICS. Therefore, when moving from a JVM pooled run time to the new JVM server run time, you see a difference in the way that you set up and control class caching.

**Note:** The JVM class cache mechanism provides benefits in terms of caching loaded classes and optimizing the just-in-time (JIT) compilation process. This provides multiple benefits as follows:

- Reduces total storage usage across several regions
- Improves Java application performance after JVM startup
- Improves JVM startup time

CICS TS V4.2 and the CICS Explorer support both JVM pool and JVM server. Therefore, you will find references, CICS commands, and displays in CICS, pertinent to the setup and control of the class cache (for JVM pooled runtime environments only). After you moved over to the new JVM servers, all those references become obsolete. For JVM server class caching, you must reference Java 6 documentation for the various commands and utilities. CICS does not customize the class cache support in any way for the JVM server.

Other than the initial setup for class caching, via the JVM server's JVM profile, you issue Java command-line commands either by the JVM profile or from a z/OS UNIX System Services shell to monitor and control your shared class cache.

By default, shared class caching is disabled. To turn on shared class caching, follow the steps in the next subsection.

### 10.5.1 Enabling the shared class cache

To enable class caching for a specific JVM server, update the associated JVM profile with the `-Xshareclasses` option and start or recycle the JVM server.

Details on setup, monitoring, and performance considerations can be found in the online information centers for Java 6, under the heading: *Class data sharing between JVMs.*
You have several suboptions available for the **-Xshareclasses** command, of which several examples are included. For complete details about the command-line options, see the following link:

http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/topic/com.ibm.jav a.doc.user.zos.60/user/classdatasharing.html

Or, issue the help option shell command: `java -Xshareclasses:help`.

Use the various suboptions to tailor the usage. For example, you can use a single class cache for each JVM server. Or, you can share one among a group of JVM servers for several CICS regions.

**Note:** This requires PM78799 to cache application classes.

Example 10-1 shows enabling class caching for a JVM server by taking all the defaults and giving the cache a unique name that is related to the associated CICS region. If this cache is shared in the future, the name would not indicate its use.

**Example 10-1  JVM profile option to enable class cache taking all the defaults**

```bash
# JVM options which enable and control class data sharing
# taking all defaults and setting a specific name
#
-Xshareclasses:name=cics.&APPLID;
#
```

Example 10-2 on page 361 shows enabling class caching for a JVM server using a shared name `cics.TskCache` and specifying a private location for the cache control files and the nonfatal option.

Using a name such as `cics.TskCache` signifies that the cache can be used by other CICS regions in the LPAR. The groupAccess option allows CICS regions running under different user IDs, but with the same primary group to share the same cache.

The nonfatal option allows the JVM to start even if the cache cannot be used.

The `cacheDir` is used to store the control files for the shared memory area. Specifying a private location can be useful if the default location `/tmp/javasharedresources` is out of space or unavailable.
The verbose option was added to receive more detailed messages. Other
verbose options exist and might result in huge amounts of output, but can be
useful for diagnosing problems.

**Example 10-2  JVM profile option to enable class taking all the defaults**

```bash
# JVM options which enable and control class data sharing
# setting a shared cache name
# setting a unique target location to isolate from other applications
# allowing more detailed status and error messages
# resetting the cache when the first region connects

-Xshareclasses:cacheDir=/u/cicsprd/java/,name=cics.TstCache,nonfatal,verbose
```

Using the utility suboptions of the **-Xshareclasses** command, you can display
information about your cache usage. Following is a list of the available utility
options of the **-Xshareclasses** command:

- **destroy**: Destroys a cache
- **destroyAll**: Tries to destroy all caches available
- **listAllCaches**: Lists all the caches
- **printStats**: Displays summary information
- **printAllStats**: Displays detailed information

Example 10-3 shows the output of the following command:

```
java -Xshareclasses:listAllCaches,cacheDir=/u/cicsprd/java/
```

Both caches from Example 10-1 on page 360 and Example 10-2 are listed.

**Example 10-3  Suboption to list all caches**

<table>
<thead>
<tr>
<th>Cache name</th>
<th>level</th>
<th>persistent</th>
<th>OS shmid</th>
<th>OS semid</th>
<th>last detach time</th>
</tr>
</thead>
<tbody>
<tr>
<td>cics.CICWKS95</td>
<td>Java6 64-bit</td>
<td>no</td>
<td>344088</td>
<td>331802</td>
<td>In use</td>
</tr>
<tr>
<td>cics.TstCache</td>
<td>Java6 64-bit</td>
<td>no</td>
<td>16409</td>
<td>4123</td>
<td>In use</td>
</tr>
</tbody>
</table>
Example 10-4 shows the output of the following command:

```
java -Xshareclasses:printStats,name=cics.TstCache,cacheDir=/u/cicsprd/java/
```

This command targets `cics.TstCache`. A separate command must be entered for each cache. The last line shows that the cache is 44% full.

**Example 10-4  Suboption to display cache statistics**

Current statistics for cache "cics.TstCache":

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared memory ID</td>
<td>16409</td>
</tr>
<tr>
<td>Cache created with:</td>
<td></td>
</tr>
<tr>
<td>-Xnolinenumbers = false</td>
<td></td>
</tr>
<tr>
<td>BCI Enabled</td>
<td>false</td>
</tr>
<tr>
<td>Cache contains only classes with line numbers</td>
<td></td>
</tr>
<tr>
<td>base address</td>
<td>0x000002004331B700</td>
</tr>
<tr>
<td>end address</td>
<td>0x0000020044300000</td>
</tr>
<tr>
<td>allocation pointer</td>
<td>0x000002004378C4D0</td>
</tr>
<tr>
<td>cache size</td>
<td>16776656</td>
</tr>
<tr>
<td>free bytes</td>
<td>8531640</td>
</tr>
<tr>
<td>ROMClass bytes</td>
<td>4656592</td>
</tr>
<tr>
<td>AOT bytes</td>
<td>938448</td>
</tr>
<tr>
<td>Reserved space for AOT bytes</td>
<td>-1</td>
</tr>
<tr>
<td>Maximum space for AOT bytes</td>
<td>-1</td>
</tr>
<tr>
<td>JIT data bytes</td>
<td>52628</td>
</tr>
<tr>
<td>Reserved space for JIT data bytes</td>
<td>-1</td>
</tr>
<tr>
<td>Maximum space for JIT data bytes</td>
<td>-1</td>
</tr>
<tr>
<td>Zip cache bytes</td>
<td>1048984</td>
</tr>
<tr>
<td>Data bytes</td>
<td>111824</td>
</tr>
<tr>
<td>Metadata bytes</td>
<td>105340</td>
</tr>
<tr>
<td>Metadata % used</td>
<td>1%</td>
</tr>
<tr>
<td>Class debug area size</td>
<td>1331200</td>
</tr>
<tr>
<td>Class debug area used bytes</td>
<td>643188</td>
</tr>
<tr>
<td>Class debug area % used</td>
<td>48%</td>
</tr>
<tr>
<td>Raw class data area size</td>
<td>0</td>
</tr>
<tr>
<td>Raw class data used bytes</td>
<td>0</td>
</tr>
<tr>
<td>Raw class data area % used</td>
<td>100%</td>
</tr>
</tbody>
</table>

# ROMClasses    = 1636
# AOT Methods   = 379
# Classpaths    = 4
# URLs          = 8
As new CICS Java bundle applications are installed for the JVM server using the cache in Example 10-4 on page 362, the cache grows in size. After installing two small application bundles, the cache size grew to 45% full.

If you defined the cache with suboption verboseIO specified, you receive output in your CICS regions dfhjvmerr file under UNIX System Services with an analysis of the class loader’s actions for each class being loaded. See Example 10-5.

Example 10-5 Output from the verboseIO option sent to dfhjvmerr

Failed to find class com/ibm/itso/sg245275/trader/ImageLoader in shared cache for class-loader id 12 with URL /u/dds0363/Java/Bundles/TraderTwoBundle/TraderTwo_1.0.0.jar. Stored class com/ibm/itso/sg245275/trader/ImageLoader in shared cache for class-loader id 12 with URL /u/dds0363/Java/Bundles/TraderTwoBundle/TraderTwo_1.0.0.jar.
Failed to find class com/ibm/itso/sg245275/trader/TraderBL in shared cache for class-loader id 12 with URL /u/dds0363/Java/Bundles/TraderTwoBundle/TraderTwo_1.0.0.jar. Stored class com/ibm/itso/sg245275/trader/TraderBL in shared cache for class-loader id 12 with URL /u/dds0363/Java/Bundles/TraderTwoBundle/TraderTwo_1.0.0.jar.
Failed to find class com/ibm/itso/sg245275/trader/TraderPL in shared cache for class-loader id 12 with URL /u/dds0363/Java/Bundles/TraderTwoBundle/TraderTwo_1.0.0.jar. Stored class com/ibm/itso/sg245275/trader/TraderPL in shared cache for class-loader id 12 with URL /u/dds0363/Java/Bundles/TraderTwoBundle/TraderTwo_1.0.0.jar.
10.5.2 Specifying cache size

As shown in Example 10-6, to specify class cache size you use the following JVM command-line parameter: `-Xscmx1M`.

*Example 10-6  JVM profile option to set the cache size*

```bash
# # JVM options which enable and control class data sharing # setting a shared cache name # setting a unique target location to isolate from other applications # allowing more detailed status and error messages # resetting the cache when the first region connects # setting the cache size to 4M #
-Xscmx4M
-Xshareclasses:cacheDir=/u/cicsprd/java/,name=cics.TstCache,verbose,reset
#
```

After recycling all the JVM servers by using cics.TstCache and rerunning a `printStats` command, the cache size is now 99% full. Creating a cache without specifying a size resulted in a 16M cache. Therefore, specifying `-Xscmx4M` lowered the cache size and was a bad choice.

*Note:* The cache size can be set only when first created. Therefore, to increase the size you must either use the reset option, or manually destroy the old cache.

10.5.3 Clean up after converting from JVM pools to JVM servers

After your migration from JVM pools over to JVM servers is complete, there are a few parameters that you can remove from your system that will not be referenced.

Remove the following JVM pool-related class cache-related SIT parameters:

- **JVMCCIZE**
- **JVMCCSTART**

You might still have **CLASSCACHE** and **CLASSPATH** parameters in your old JVM pool-related JVM profiles, but those should not be in use anymore and also can be deleted.
Chapter 10. Migration

10.6 Thread-safe Java applications

With CICS TS V4.2 and the newly introduced JVM server, all Java applications that run in a JVM server now must consider the concept of concurrency or thread-safe programming concepts. This is not a thread-safe book. Therefore, the concepts and programming techniques for thread-safe programming are not described.

What is important is that you understand that the new JVM server environment gives your application the ability to run concurrently on multiple T8 TCBs at the same time. Your CICS Java programs might execute code on several threads simultaneously in a shared address space where each of those threads has access to virtually all of the memory of every other thread.

Additionally, in a CICSPlex environment with multiple CICS regions running different copies of your application, the sysplex running the CICS regions can provide access to shared resources stored in the coupling facility. For example, shared temporary storage queues.

Your applications must be written to account for access to shared resources and provide the appropriate locking when updating shared resources.

The CICS JVM Application Isolation Utility can be used to audit the use of static variables in your Java applications. For more information, see the following link:

Example 10-7 shows the output of executing the following command:

```
DFHIsoUtil /user/cics/cicsdemo/classes/Trader/TraderTwoBundle_1.0.0/TraderTwo_1.0.0.jar
```

You can add the -verbose option, but the output was too large to include here.

Example 10-7  Output from the Isolation Utility against the TRADER2 application

CicsIsoUtil: CICS JVM Application Isolation Utility
Reading jar file:
/user/cics/cicsdemo/classes/Trader/TraderTwoBundle_1.0.0/TraderTwo_1.0.0.jar

Number of classes inspected: 16
End of jar file:
/user/cics/cicsdemo/classes/Trader/TraderTwoBundle_1.0.0/TraderTwo_1.0.0.jar

Number of jar files inspected : 1
Number of class files inspected : 0

No static writes were found in the analysis, meaning that all the static variables found were not updated elsewhere in the code.

Use of the Isolation Utility does not mean that your application is threadsafe. The utility inspects Java bytecodes and reports on the static variables that are used by each class. Ensure that your application is not modifying any static variables.
Installing the CICS Explorer software development kit

The base development environment, for Java programs that are developed for CICS, is the *Eclipse Integrated Development Environment (IDE)*. Eclipse is an open development platform, produced by the Eclipse Foundation (www.eclipse.org), composed of extensible frameworks, tools, and run times for building, deploying, and managing software across the lifecycle.

In this appendix, we provide instruction on downloading and installing both the Eclipse IDE and the CICS Explorer software development kit (SDK).
Installing Eclipse

The development environment for Java programs in CICS is the CICS Explorer SDK, which is required to run as a plug-in to an Eclipse IDE. The first step in installing the CICS Explorer SDK is to install Eclipse onto a workstation. The following CICS Explorer website gives details of the required levels of Eclipse that must be installed:


At the time of writing, CICS Explorer SDK V1.1.1 requires Eclipse V3.6.2. A link for this level of the SDK can be found at the following website:


Version 3.6.2 is the version of Eclipse that we used to build the examples in this book. See Figure A-1.

Figure A-1  The build of Eclipse used in this book
Installing the CICS Explorer SDK

The CICS Explorer SDK is an Eclipse-based framework for developing extensions to the CICS Explorer. It also provides support for developing Java applications to run in any supported release of CICS. It provides support for Java class library for CICS (JCICS) and for packaging applications to comply with the OSGi specifications.

Now that you have Eclipse installed, you must download and then install the CICS Explorer SDK into Eclipse. The CICS Explorer SDK can also be installed into Rational Application Developer for WebSphere Software using the IBM Installation Manager. The SDK is also provided as part of Rational Application Developer for System z. We do not cover these scenarios in this book.

Downloading the Explorer SDK

Using a web browser, go to the CICS home page at http://www.ibm.com/cics. From this page, scroll down and click the CICS Explorer link that is shown in Figure A-2.

![Figure A-2  CICS Explorer link](image)

Then, click the download Explorer icon, as shown in Figure A-3.
On the next page, scroll down and use the **Download site** link. On the next page, either enter your IBM ID and password if you have one, or click the **Proceed without an IBM ID** link.

You are then presented with a list of download options. Select the CICS Explorer option, as shown here in Figure A-4.

![Figure A-4 CICS Explorer download selection](image)

Then, perform the following actions:

1. Click **Continue** at the bottom of the page.
2. On the next page, read and agree to the license information presented and select which type of communication you want to receive from IBM.
3. Click **I Confirm**.
4. The final page now presents a list of CICS Explorer downloads. They can either be downloaded by using **Download Director** or via a standard HTTP download. Choose one by clicking the appropriate tab and then click the link for **CICS Explorer**.
5. We chose to do this using HTTP, so we clicked **Download using http** at the top and then clicked **Download now** against **CICS Explorer SDK**, as shown in Figure A-5, not CICS Explorer SDK (Installation Manager repository).

![Figure A-5 Download now](image)
Installing the Explorer SDK

Start Eclipse if you have not already started it. Close the welcome panel, if it loaded. Then, select Help → Install New Software, as shown in Figure A-6.

![Install New Software panel]

Figure A-6   Install New Software panel
This selection opens the Available Software window, as shown in Figure A-7, from where you can install the SDK compressed file from the saved file on your workstation.

**Figure A-7**  Available Software installation window

Click **Add**, which opens the Add Repository window (Figure A-8.)

**Figure A-8**  Add repository

- In the **Name** field, enter a name for the repository that you are about to add. For example, CICS SDK.
- Then, click **Archive** and go to the compressed file that you saved in “Downloading the Explorer SDK” on page 369. Select this and click **OK**.
- The installation window then updates to show the CICS SDK components that can be installed into Eclipse, which can be seen in Figure A-9.

![Install](image)

*Figure A-9  Available CICS SDK components*

- Click **Select All** followed by **Next** and Eclipse installs the components into the Eclipse IDE environment.
  Eclipse can now be used to develop Java applications for deployment into CICS Transaction Server.
Java EE Connector Architecture tools

Java EE Connector Architecture (J2C) tools are provided in Rational Application Developer and Rational Developer for System z as an optional feature. You can use the J2C wizards to easily generate Java data binding classes that map to COBOL, C, or PL/I data structures. In addition, the J2C tools provide the ability to create J2C beans that can be used together with the CICS Transaction Gateway JCA resource adapter to encapsulate interactions with CICS.

In the following sections, we show how to generate data binding classes using the J2C wizards in with Rational Developer for System z V8.0.3, which can then be deployed within a CICS Java virtual machine (JVM) server. The example is based on the GENAPP CICS sample application, which is used in Chapter 8, “Interacting with structured data” on page 253.
Rational Application Developer

Rational Application Developer is an industry-leading IDE for developing Java and J2EE-based applications. It is built on the Eclipse platform and extends the base functionality by using plug-ins. The easy-to-use wizards and templates generate high-quality, resilient, scalable applications that are optimized to run on WebSphere Application Server and IBM WebSphere Portal Server products. Rational Application Developer provides the J2C wizards as an optional feature.

Rational Developer for System z is an IDE for developing Java and traditional mainframe applications such as COBOL or PL/I. Rational Developer for System z consists of a common workbench and an integrated set of tools that support end-to-end, model-based development, runtime testing, and rapid deployment of applications. Like Rational Application Developer, it is built on the Eclipse platform and extends the base Eclipse functionality through the use of plug-ins. It offers an IDE with advanced, easy-to-use tools and features to help both WebSphere Application Server, CICS, and IMS developers rapidly design, code, and deploy complex applications.

Java EE Connector Architecture tools installation

Rational Developer for System z is the recommended IDE for the development in CICS. Therefore, in this section, we use Rational Developer for System z V8.0.3 as the example to show how to install J2C tools, although we could also have used Rational Application Developer to generate the same Data Binding beans.
The J2C tools are an optional feature of Rational Application Developer so first we had to install the Rational Application Developer for WebSphere Software component as shown in Figure B-1.

Figure B-1   Installation Rational Application Developer component with Rational Developer for System z
When the IBM Installation Manager shows the features to install, it is necessary to select the **Java EE Connector (J2C) development tools** feature, as shown in Figure B-2.

![IBM Installation Manager](image)

*Figure B-2  J2C feature during installation*

If Rational Developer for System z was previously installed on the workstation, ensure that the J2C feature was also installed. To verify the installed features after starting Rational Developer for System z, click **Help → About IBM Rational Developer for System z with Java → Installation Details → Plug-ins.**
You will see a list of installed plug-ins for J2C tools, as shown in Figure B-3. If they are not present, install the J2C feature using the IBM Installation Manager as shown in this section.

![IBM Rational Developer for System z with Java Installation Details](image)

**Figure B-3** Plug-ins for J2C tools

**Generate data binding classes**

Data binding classes are Java classes encapsulating COBOL, PL/I, or C language structures. They can be generated in Rational Application Developer or Rational Developer for System z using an importer wizard, and used as input or output types for interactions with CICS either as stand-alone classes or together with the generated J2C beans.
In this section, we generate the Java classes from the COBOL data structure defined by GENAPP to describe customer and policy information as follows.

1. Click **File → New → Other → J2C → CICS/IMS Java Data Binding**, as shown in Figure B-4. This starts the CICS Java Data Binding Wizard.

![Figure B-4 J2C Java Data Binding Wizard](image-url)
2. Click **Next** to open the Data Import window (Figure B-5).

![Figure B-5 J2C Data Binding Wizard: Data Import](image)

The J2C tools support the mapping of various language structures. We require the conversion of COBOL data structures to their Java equivalent. Therefore, in the *Choose mapping* drop-down list, select **COBOL to Java**. To create the Java equivalent of a COBOL data structure, the wizard must know how the COBOL data structure is defined. Typically, this structure is available within the COBOL program file or in a separate COBOL copybook. In the **COBOL file** text box, specify the COBOL program or COBOL copy book where these definitions are available. By default, J2C tools assume that any file with the `.cbl` extension is a complete COBOL program, and any file with the `.cpy` extension contains only the data structure definitions.

3. These assumptions can be altered by changing the preferences for the COBOL file importer by clicking **Window → Preferences**.
4. In the Preferences panel (Figure B-6), expand **Importer**, and select **COBOL**. The General tab provides options to select the target platform encoding, byte order, and so forth. In this example, we select the target platform as z/OS.

Specify all the system-specific parameters in the Preferences window before proceeding to the next step. The J2C tool expects a syntactically complete and correct COBOL structure. If the structure contains compiler-specific key
words or missing sections, this might cause errors when attempting to parse the structure.

5. Click **Next** to choose the appropriate data structure, as shown in Figure B-7.

The default selections presented for the platform and code page are based on the choices made in the previous step. Change these selections if necessary. In this example, the COBOL file LGPOLICY.cbl that we imported contained six level one data structures, as shown in Figure B-7. These structures need to be converted to their Java equivalents. Select this and proceed to the next step.

![Figure B-7 Selecting the COBOL data structures](image)
6. Click **Next** to modify the Saving Properties options (Figure B-8).

   By default, the binding Java class is given a name that resembles the data structure identifier. In this example, the default Java class name that is given by the wizard is `DB2CUSTOMER`. We add this binding class to the `com.ibm.itso` package, which belongs to the project itso.

![Import](image)

**Figure B-8** Saving Properties options

7. Click **Finish**. The wizard generates the DB2CUSTOMER Java class. We use the same way to generate the Java classes for the rest of the COBOL data structures.

**Note:** To use the generated J2C data bindings at run time in the CICS Open Services Gateway initiative (OSGi) JVM server environment, it is necessary to wrap the J2C `marshall.jar` file into the deployed OSGi bundle. For an example of how to do this, see “Wrapping JARs as plug-in projects” on page 274.
Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

The following IBM Redbooks publications provide additional information about the topic in this document. Note that some publications referenced in this list might be available in softcopy only.

- Java Application Development for CICS, SG24-5275-03
- Developing Connector Applications for CICS, SG24-7714-00
- Introduction to CICS Dynamic Scripting, SG24-7924-00
- Threadsafe Considerations for CICS, SG24-6351-04

You can search for, view, download, or order these documents and other Redbooks, Redpapers, Web Docs, draft, and additional materials, at the following website:

ibm.com/redbooks

Other publications

This publication is also relevant as a further information source:

CICS Transaction Server Application Architecture, TIPS0922

Online resources

These websites are also relevant as further information sources:

- Open Services Gateway initiative
  http://www.osgi.org/
- CICS Explorer downloads
Help from IBM

IBM Support and downloads
ibm.com/support

IBM Global Services
ibm.com/services
CICS and the JVM server: Developing and Deploying Java applications
This IBM Redbooks publication provides information about the new Java virtual machine (JVM) server technology in IBM CICS Transaction Server for z/OS V4.2. We begin by outlining the many advantages of its multi-threaded operation over the pooled JVM function of earlier releases. The Open Services Gateway initiative (OSGi) is described and we highlight the benefits OSGi brings to both development and deployment.

Details are then provided about how to configure and use the new JVM server environment. Examples are included of the deployment process, which takes a Java application from the workstation Eclipse integrated development environment (IDE) with the IBM CICS Explorer software development kit (SDK) plug-in, through the various stages up to execution in a stand-alone CICS region and an IBM CICSPlex environment.

The book continues with a comparison between traditional CICS programming, and CICS programming from Java.

Further chapters are provided to demonstrate interaction with structured data such as copybooks, and how to access relational databases by using Java Database Connectivity (JDBC) and Structured Query Language for Java (SQLJ).

Finally, we devote a chapter to the migration of applications from the pooled JVM model to the new JVM server run time.