IBM

New Ways of Running Batch Applications on z/OS

Volume 4 IBM IMS

Technology overview

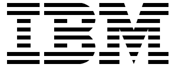
Modernization options

Samples

Denis Gaebler
Alex Louwe Kooijmans
Elsie Ramos

Redbooks

ibm.com/redbooks



International Technical Support Organization

New Ways of Running Batch Applications on z/OS: Volume 4 IBM IMS

May 2014

SG24-8119-00

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

First Edition (May 2014)

This edition applies to the following software levels:

- z/OS Version 1 Release 12 and Release 13
- IBM 64-bit SDK for z/OS, Java Technology Edition, V6

© Copyright International Business Machines Corporation 2014. All rights reserved.
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices v
Trademarks vi

Preface vii
Authors vii
Now you can become a published author, too! viii
Comments welcome viii
Stay connected to IBM Redbooks viii

Chapter 1. Modernizing IMS batch 1
1.1 Introduction 2
1.2 IMS defined 2
1.3 Modern view of IMS batch processing 3

Chapter 2. Implementation of IBM IMS batch applications in Java 7
2.1 Software prerequisites 8
2.1.1 Minimum software levels 8
2.2 Use of development environment 8
2.3 How to write and test IMS Java applications outside IMS 9
2.4 Java frameworks used with IMS Java 9
2.4.1 Using Hibernate as Object Relational mapper with IMS DB 10
2.4.2 Java Persistence API 10
2.4.3 Using Spring in Java parts of IMS applications 10
2.5 Access of IMS Java batch applications with pureQuery 11
2.6 JVM tuning considerations 12
2.7 Debugging Java applications in IMS 12
2.7.1 Debugging a Java BMP region 12
2.8 Diagnostics and monitoring of Java in an IMS environment 17
2.8.1 IBM Monitoring and Diagnostic Tools for Java - Health Center 17
2.8.2 Rational Agent Controller, Rational profiling, and Healthcenter Plug-in 18
2.8.3 JConsole 19
2.8.4 IBM HeapAnalyzer 20
2.8.5 Profiling applications 23
2.8.6 Monitoring JVMs in IMS regions 23
2.9 Java interoperability with COBOL in IMS batch applications 24
2.9.1 How Java can call COBOL and vice versa 24
2.9.2 JNI calls using COBOL INVOKE 27
2.9.3 COBOL code evolution 27
2.9.4 Latest COBOL INVOKE versus JNI API measurements 34
2.9.5 JNI programming considerations 35
2.9.6 Options to pass data items between COBOL and Java 35
2.10 Generating Java classes 35
2.10.1 J2C wizards 35
2.10.2 JZOS Record Generator 36
2.11 Restrictions for COBOL Java interoperability 37
2.12 Abend and error handling 37
2.12.1 Alternate options 38
2.13 z/OS considerations 38

© Copyright IBM Corp. 2014. All rights reserved.

Chapter 3. Mixed language applications . . . 41
3.1 Accessing DB2 from mixed language applications . . . 42
3.2 Accessing WebSphere MQ from mixed language applications . . . 43
3.3 Debugging mixed language applications . . . 44
3.3.1 Tools to debug mixed language applications . . . 45
3.4 IMS preload in a mixed environment . . . 48

Chapter 4. Alternate processing options . . . 51
4.1 IMS callout to external services . . . 52
4.1.1 Synchronous calls . . . 52
4.1.2 Asynchronous calls . . . 53
4.1.3 Both synchronous and asynchronous callout . . . 53
4.1.4 Call DB2 Stored Procedures from the application . . . 54
4.1.5 WebSphere Transformation Extender . . . 54
4.1.6 WebSphere z/OS Optimized Local Adapters . . . 54
4.2 Calling IMS transactions from traditional batch . . . 57
4.2.1 Writing a COBOL client . . . 58
4.2.2 The OTMA Callable Interface . . . 58
4.2.3 Use of DB2 stored procedures . . . 60
4.2.4 WebSphere Transformation Extender . . . 60
4.2.5 WebSphere MQ with its IMS OTMA Bridge . . . 61
4.3 Accessing IMS data as result sets from traditional batch . . . 61
4.3.1 Using Business Rules Engines from batch . . . 61
4.3.2 Speeding up long running DB2 queries . . . 62
4.3.3 Speeding up calls to SQL-only DB2 stored procedures . . . 63
4.3.4 Using data transformations in batch . . . 63
4.4 Best practices for small batches . . . 66
4.4.1 Reduce the overhead of JVM startup . . . 66
4.5 Summary . . . 67

Chapter 5. IMS batch samples . . . 69
5.1 Sample IMS Java batch program . . . 70
5.1.1 Software used in our environment . . . 70
5.1.2 Procedures used in our environment . . . 70
5.1.3 Configuration used in our environment . . . 71
5.2 Sample Java configuration and IMS BMP calls . . . 73
5.2.1 Sample Java configuration for an IMS Batch Message Program . . . 73
5.2.2 Sample application IMS BMP COBOL calls Java . . . 74
5.2.3 Sample application IMS BMP PL/I calls a Java method . . . 77
5.2.4 Sample application IMS Java Batch Program Java calls COBOL . . . 78
5.3 Sample Java frameworks used with IMS Java . . . 82
5.3.1 How to install all required plug-ins into Rational Developer for System z . . . 83
5.3.2 Download the Hibernate JARs into the sample workspace . . . 83
5.3.3 Hibernate as Object Relational mapper with IMS DB . . . 84
5.3.4 Using JPA as OR mapper with DB2 . . . 98
5.4 Summary . . . 102

Related publications . . . 103
IBM Redbooks . . . 103
Online resources . . . 103
Help from IBM . . . 104

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:

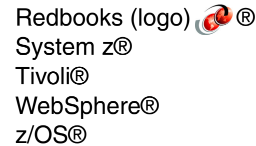
This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing 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.

© Copyright IBM Corp. 2014. All rights reserved.

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:

CICS Explorer®
CICS®
DB2®
developerWorks®
IBM®
IMS™
Language Environment®
Optim™
pureQuery®
Rational Team Concert™
Rational®
Redbooks®
Redbooks (logo)  ®
System z®
Tivoli®
WebSphere®
z/OS®

The following terms are trademarks of other companies:

Netezza, and N logo are trademarks or registered trademarks of IBM International Group B.V., an IBM Company.

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

Mainframe computers play a central role in the daily operations of many of the world's largest corporations. Batch processing is still a fundamental, mission-critical component of the workloads that run on the mainframe. A large portion of the workload on IBM® z/OS® systems is processed in batch mode.

This IBM Redbooks® publication is the fourth volume in a series of four. They address new technologies introduced by IBM to facilitate the use of hybrid batch applications that combine the best aspects of Java and procedural programming languages such as COBOL. This volume focuses on the latest enhancements in IBM Information Management System (IMS™) batch support. IMS has been available to clients for 45 years as IMS Transaction Manager, IMS Database Manager, or both.

The audience for this book includes IT architects and application developers with a focus on batch processing on the z/OS platform.

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.

Denis Gaebler is a Technical Sales Specialist at IBM in Germany. He holds a degree in business with a specialization in computer science from Staatliche Studienakademie Dresden. He has been working in the areas of IBM IMS and IMS Web Enablement since 1997. His areas of expertise include IBM WebSphere® on z/OS, IMS DB and TM, Service-Oriented Architectures, and Enterprise Application Integration. Currently, he is working with IMS Java, IMS connectivity solutions, COBOL and Java Integration, application servers, and Eclipse-based application development tools.

Alex Louwe Kooijmans is a Senior Architect at the Financial Services Center of Excellence at IBM Systems and Technology Group. Before this position, he spent almost 10 years in the International Technical Support Organization leading IBM Redbooks projects, teaching workshops, and running technical events with a focus on using the IBM mainframe in new ways. Alex also worked as Client Technical Advisor to various banks in the Netherlands and performed various job roles in application development. His current focus is on modernizing core banking systems and the role of the IBM current mainframe technology.

Elsie Ramos is a Project Leader at the ITSO, Poughkeepsie Center. She has over 30 years of experience in IT, supporting various platforms including IBM System z® servers.

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

Rich Conway
Michael Schwartz
IBM International Technical Support Organization, Poughkeepsie Center

Gary Puchkoff
Software Architect, IBM Systems and Technology Group, z/OS New Technology
Poughkeepsie, US

© Copyright IBM Corp. 2014. All rights reserved.

Gary Wicks
IBM Canada

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

- Send your comments in an email to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

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:

  https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm

- Stay current on recent Redbooks publications with RSS Feeds:

http://www.redbooks.ibm.com/rss.html

1

Modernizing IMS batch

In this chapter, we describe new modernization options as a result of the changes in the architecture of existing and new Information Management System (IMS) based batch applications. There is a brief review of IMS as a batch container followed by a description of why it is a good idea to modernize within the IMS container.

© Copyright IBM Corp. 2014. All rights reserved.

1.1 Introduction

There are many ways to modernize IMS batch while still running existing workloads uninterrupted.

- Some clients rewrite their applications in existing languages using new agile development techniques.
- Others implement new ways of performing interactions using rules engines or initiating a callout to external services.
- To avoid large migrations, a continuous modernization can be performed while still running the applications in the IMS container.
- Many clients already have batch applications that are based on IMS and they use the key feature of checkpoint restart functionality for advanced control over their applications.

1.2 IMS defined

IMS consists of a transaction manager and a database manager:

- Transaction manager
  - Ensures that a transactional or batch unit of work conforms to the ACID properties (atomicity, consistency, isolation, and durability). Includes commit and rollback with Resource Managers such as IMS databases, IBM DB2® databases, and WebSphere MQ.
  - Provides checkpoint restart support for batch applications in order to be able to continue later from a point of error (in most cases full output data sets, and so on) and provides the ability to save a piece of storage (workarea) that saves the variables at checkpoint time and presents it to the application at restart time, so it can continue, as if it was running uninterrupted.
  - Provides sync point management for external resources like IBM DB2 for z/OS databases and WebSphere MQ for z/OS and allows these external subsystems to be part of the IMS managed unit of work. This includes the ability to commit or rollback changes of resources by using a two-phase commit capable protocol.
  - Provides Queue Manager functionality, through the use of message queuing through a dedicated set of message queue buffers and data sets aligned with a single IMS or Shared Queues between members of an IMSplex.
  - Allows access to IMS Transactions from many types of environments running on z/OS like DB2 Stored Procedures, Resource Recovery Services (RRS) controlled Batch Jobs, IBM CICS® and WebSphere Application Server and from distributed applications through IMS Connect and WebSphere MQ.
- Database manager
  - Manages hierarchical IMS databases and ensures integrity while IMS applications access the data in parallel.
  - Allows access to IMS databases from many types of environment running on z/OS like DB2 Stored Procedures, RRS controlled Batch Jobs, CICS and WebSphere Application Server, and from distributed applications through IMS Connect.
  - Provides a batch container in stand-alone mode (known as *Database Control environment*—DBCTL) like the one included in IMS Transaction Manager. Therefore, it benefits from the same checkpoint and restart support.

Historically, IMS has been providing for 45 years a stand-alone batch container running the IMS subsystem entirely in the batch address space without the need to connect to an IMS control region. The DL/I or DBB Batch regions do not share IMS databases with online IMS subsystems unless IMS sysplex data sharing is implemented.

Today, most of the users run “online batch” jobs as privileged for application workloads due to operational and business requirements. “Offline batch” jobs are still used for utilities and for application workload that were not able to be changed to implement intermediary checkpoint/restart logic to accept parallel processing. Therefore, when we mention “batch” in this chapter, it refers to online batch variations, such as IMS batch message processing (BMP) programs, IMS message-driven BMPs, and IMS Java batch programs (JBPs).

1.3 Modern view of IMS batch processing

IMS batch processing has been available for 45 years. Batch is clearly one of the main features of IMS. It offers checkpoint restart capabilities so that applications do not have to be restarted from the beginning after failures. The programmer has very little to do to implement it because the IMS subsystem manages everything under the covers. In a failure, applications are automatically restarted to the latest checkpoint. The modernization of batch in IMS might be the best option when there is a requirement to call services that are off-platform or Java based.

To update IMS based batch to use COBOL and Java interoperability, the following points need to be considered:

- Should the current JCL be changed or is the current tooling capable of generating something different to describe the batch work?
- Should the current static specification of input and output data sets by JCL be changed?
- Should the current way of checkpoint writing (considering tools that allow the specification of the checkpoint frequency dynamically or at job/step start) be changed?
- Should the current way of checkpoint restart control (IMS based writing of checkpoints) be changed?
- Should the batch modernization platform be a different one from the current online transaction processing (OLTP) and batch processing in use?
- Should the current accounting based on SMF30 records be changed to use some other SMF records? SMF30 records also contain reporting about the System z Integrated Information Processor (zIIP) and System z Application Assist Processor (zAAP) usage.
- Should there be an incremental conversion of COBOL to Java?
- Should there be different tooling to monitor, debug, and report runtime errors as being introduced with heterogeneous batch environments that allows for a simple choice for running batch on other platforms?
- Should the current security solutions and procedures to authorize batch in the z/OS environment be changed?
- Should the same tools for auditing and review (IMS logs) be used?
- Should the same tools for coordinated recovery be used?
- Should deadlock behavior when online and batch workload compete for the same resources be replaced with a new solution? This includes deadlock detection, reports, and analysis sources.

When the answer to most of these questions is “no”, then staying with IMS for batch modernization is likely the better alternative.

The following solutions are possible with little overhead for IMS based solutions:

- Consolidation of batch or Java based platform programs can be started as IMS Java batch by having a main method that is started by IMS. The mapping is between the Java class name and the program specification block (PSB).
- Tight integration with IBM Tivoli® Workload Scheduler. This is workload automation software that uses business policies to support business goals.
- All JVM functions

  Includes but is not limited to JZOS Record Generator, JZOS Record IO, calling remote Enterprise JavaBeans (EJB beans) through RMI/IIOP, and rules engines such as IBM Operational Decision Manager.
- Frameworks such as Hibernate, Spring, and Java Persistence API (JPA) can be used in IMS applications and in Java methods that are parts of traditional IMS applications.
- Basically everything that can run as Java main or plain old Java object (POJO) or Java 2 Platform, Standard Edition (J2SE) applications can run easily with few or no code changes as an IMS Java application or can be called from within an existing IMS application.
- With COBOL Java interoperability, the same COBOL/Java modules/methods and call sequences can be easily reused between IMS online and batch.
- When running in traditional IMS regions (not Java dependent regions) the entire application uses the IMS preload and preinit functionality, which is not available with other solutions.
- DB2 quiesce and IMS DB quiesce allow the creation of coordinated consistency points for database recovery without the need to stop or abend the batch work.
- IMS uses the same JCL patterns that have been in place. This includes the generation of JCL and data definition (DD) statements by tooling on the mainframe.
- IMS does not require additional software to run Java batch or batch that mixes COBOL and Java. Keep in mind that access to IMS DB and DB2 is always transactional, local, and within the same unit of work.
- Batch with IMS allows you to stay with the existing job network, monitoring, debug, and problem determination tooling. This includes the same tooling for both online and batch workloads.
- Java workload running in IMS is eligible to run on zAAPs or zIIPs, depending on the configuration: zAAPs available or zAAP on zIIP enabled, to the same extent as all other Java workloads on z/OS.
- Deadlocks are usually a design problem or a timing/scheduling problem. If they occur frequently for the same workload, the application should be investigated. In the case of deadlocks between online and batch workloads, IMS allows for the IMS transaction to be rolled back, requeued, and executed later, when there is a better chance that the batch job has already checkpointed and the competing resources have been freed up:
  - While a batch job can also abend with a U777 abend, automation can be set up to restart the job with the latest checkpoint.
  - There are also tools on the market that can adjust the checkpoint frequency based on transaction rate, time, CPU usage, or other criteria to allow the batch workloads to continue without user interaction.
  - Again, deadlocks that occur frequently in the same workload should be investigated.

- The IMS queues can be used as intermediate storage for messages to drive other message-driven batch applications, or to drive IMS online applications as part of the overall data and application workflow. There are clients who are using IMS queues for that purpose, and clients that use WebSphere MQ or DB tables as queued storage.

The overall consideration of which function, software, or architectures to use is not trivial. There are so many capabilities in IMS that the decision to “rip and replace” or “reuse and extend” IMS batch logic should be implemented while keeping operational and performance constraints in mind. Let the business, operational, and performance constraints influence the decision. This book can help to reuse and extend existing IMS batch applications and scenarios.

2

Implementation of IBM IMS batch applications in Java

In this chapter, we describe the option to replace existing COBOL batch applications with a pure Java implementation. The Information Management System (IMS) container for this purpose offers the Java Batch Programs (JBPs). These applications run in IMS regions that are configured to start a Java virtual machine (JVM) and start the Java program's main method.

Topics covered in this chapter include:

- Software requirements and the use of the development environment
- Running Java applications in IMS and how to interface traditional languages with Java
  - How to use persistence frameworks like Hibernate and Java Persistence API (JPA)
  - How to use the IBM pureQuery® function for Java access
- Error handling and z/OS considerations

© Copyright IBM Corp. 2014. All rights reserved.

2.1 Software prerequisites

Java language interoperability works with IMS Version 10 and higher. However, there are some options that are only available in IMS V11 or IMS V12. Unless otherwise noted, all samples and configurations in this book work with *IMS V11*. At the time of writing this book, no dependencies were found where the solutions could not be implemented with an earlier supported release or version of z/OS software.

2.1.1 Minimum software levels

The minimum software levels for interoperability with Java are as follows:

- All supported levels of z/OS
- All supported levels of IMS
- All supported levels of DB2 for z/OS

  APAR PQ74629 for Java application programs that access DB2 for z/OS subsystems from JMP or for JBP regions that must be applied to the DB2 for z/OS subsystem
- All supported levels of JDK for z/OS
- All supported level of WebSphere MQ for z/OS
- Enterprise COBOL for z/OS V4.2 (minimum for JDK 5 for z/OS or higher)
- Enterprise PL/I for z/OS V3.4
- IBM Rational® Developer for System z V8.0.3
- IBM Debug Tool V12

Some features that are described in this chapter require higher levels of software. For example, certain JDK options are only available on JDK 6 or JDK 6.0.1 for z/OS. Also, additional functions for handling the loading and unloading of modules in IMS regions are only available in maintenance for IMS V11 and IMS V12; they will not be retrofitted to IMS V10.

Make sure that you apply the latest program temporary fix (PTF) levels to the software stack.

2.2 Use of development environment

In many cases, the development and versioning for Java applications is different from what is used, for example, in mainframe COBOL applications.

There are different approaches that can be used when developing Java applications:

- Some mainframe vendors have already improved or enhanced their products so that they can also be used from the Eclipse environment.
- Support of the development, build, and deployment of Java based applications either by integration with the existing infrastructure or separate and running in parallel to the existing non Java build infrastructure.
- Use of the existing Java development environments and its build and versioning infrastructure. For example, the IBM Rational Team Concert™ can be used from Rational Developer for System z. At client sites, Eclipse-based development environments with Maven builds have been used in addition to Concurrent Versions System (CVS) or subversion as versioning and team infrastructure.

In summary, there are software products from IBM, vendors, and open source, plus tools that can suit the needs of development, team, versioning, and building the infrastructure for enterprise use. The approach, requirements, and features should be carefully evaluated for your specific environment.

2.3 How to write and test IMS Java applications outside IMS

Today, Java application programmers prefer to do programming and testing in an integrated environment. In this section, we discuss some hints and tips about how to write IMS applications in an environment, such as Eclipse, which allows for testing without the need to run on z/OS in an IMS region and includes database access to IMS DB and DB2.

The simplest approach is to treat the future IMS Java application as a Java main program or plain old Java object (POJO). If the application is runnable outside a Java Platform, Enterprise Edition 2 (Java EE 2) application server, and IMS with no message queue access and no system calls, you can set up remote connections to IMS DB using IMS Open Database Manager or you can use the Java Database Connectivity (JDBC) drivers on DB2 on z/OS. For further details, refer to *IMS 11 Open Database*, SG24-7856.

With this approach, programming and testing can be done in Eclipse, and the integration tests are usually done on the mainframe. When finished, the connection URL for IMS DB and DB2 changes from running inside an IMS region and the code to access the IMS message queue (getUnique + loop around getNext + insert). This approach might not fit all scenarios, but data access and data manipulation services can be developed and tested without the need to write and deploy to z/OS and to run as an IMS region on z/OS.

Integration testing, such as the interaction with other batch workloads or with the IMS message queue (queuing messages or triggering transactions), still needs to be done on the mainframe with z/OS and IMS.

2.4 Java frameworks used with IMS Java

When programming only in Java, most clients plan to use Java frameworks that have special implementations, for example object relational mappers. Here is a list of some frameworks that have worked well at client sites:

- Hibernate with DB2 and IMS DB (IMS JDBC Drivers)
- Java Persistence API (JPA) with DB2
- Spring
- IBM Operational Decision Manager JRules J2SE Rule Execution Server
- JavaMail packages
- RMI-IIOP Client to call remote EJB beans
- ISV Software with J2SE interfaces
- External WebServices by using generated Web Service Client code (for example, Apache AXIS)

In this section, we review how to use some of the Java frameworks: Hibernate, JPA, and Spring in the Java part of IMS applications.

Refer to samples for 5.3.3, “Hibernate as Object Relational mapper with IMS DB” on page 84 and 5.3.4, “Using JPA as OR mapper with DB2” on page 98.

Note: We do not show samples for every possible implementation.

2.4.1 Using Hibernate as Object Relational mapper with IMS DB

Hibernate is a Java Persistence tool that allows you to map objects against databases. You can use the IMS Open Database feature to leverage your existing databases with Hibernate.

> **Restriction:** There is little or no tooling available that can assist in the creation of Java class objects that adhere to Hibernate conventions (class names, method names, and so on) besides the IMS Enterprise Suite Explorer. It is used to map inputs from the database description (DBD), program specification block (PSB), and copybooks into Java metadata,
>
> Therefore, much of the work that we describe to map an existing IMS DB to an object hierarchy must be done manually.

IMS environment features for Hibernate

IMS is a transaction manager and IMS transactions are committed implicitly upon successful execution. IMS does not support user transactions or multiple commits within an IMS unit of work. IMS maintains the connections to IMS and DB2.

The configuration used for Hibernate should use Java Transaction API (JTA), which is configured to wrap the IMS transaction manager functionality. Hibernate caching can be turned on. It might also help to reduce access to frequently used objects. There is a distinction between cache for a single JVM and cache for a cluster. The use of Hibernate connection pooling and transaction management can produce unpredictable results.

Because the Hibernate session can be configured as static, it is possible for the Hibernate cache to persist for the whole lifetime of the JVM. Hibernate caching is not similar to IMS DB or DB2 database caches. The invalidation in the Hibernate cache is based on Time to Live specified in the configuration, so for the application the object from the cache might be outdated. It is possible to define certain objects so that they are always reloaded from the database during access.

The use of Hibernate built-in cache with *EhCache* as the Hibernate second-level cache works on z/OS. EhCache also provides a feature, where a separate JVM can be configured as a cache server. The IMS batch jobs using Hibernate can then connect to this JVM. Performance tests have been done, but they vary depending on the access patterns, so it is best to test the specific application access patterns that will be used.

2.4.2 Java Persistence API

Java Persistence API (JPA) is a simplified programming model that is used for object relational mapping and data persistence. Data persistence makes sure that applications that retrieve and update data are kept in sync with the current state of the back-end database. Traditionally, this was done using Java Database Connectivity (JDBC) APIs and other data frameworks. However, this usually involved writing complex query statements to add or modify data.

JPA simplifies this process by using Java representations of the database tables and providing a set of APIs to persist and query your data.

2.4.3 Using Spring in Java parts of IMS applications

Clients have also successfully used Spring in the Java parts of IMS applications.

There is a Rational Developer for System z project, where a Java main method is used to execute the sample. An Ant script can be used to build and upload the required JAR file to run the sample. Then, put in a class path and change the DFSJVMAP member.

2.5 Access of IMS Java batch applications with pureQuery

IMS applications can connect to DB2 on z/OS for database access. For Java access, there is the *pureQuery* function for DB2 on z/OS. This has the benefit of allowing the capture of dynamic JDBC calls that can be turned into a package with static SQL. pureQuery also works when running Java in any IMS region. From the IMS side, the pureQuery JAR files need to be in the class path of the JVM running in the IMS region.

pureQuery allows dynamic JDBC applications to use statically bound packages. This has the benefit that the DB2 admin can use a single authorization environment where static SQL is authorized for the package and dynamic SQL is authorized for the object. Requiring only one authorization for a single application is a benefit when using mixed language applications that interoperate with Java.

It is possible to capture the dynamic JDBC calls along with most parts of the application, then run it static. Then, define the additional dynamic JDBC statements that have not been recorded during the capture phase and should be executed, rejected, or recorded and later bound to the static package, the incremental recording.

pureQuery can also perform the following functions:

- Fix dirty programming, for example, to not use parameter markers for Where clauses, which cause the optimizer to calculate the path for every SQL statement.
- Allow you to bind a WHERE abc=? SQL to the package and execute all WHERE=value statements that have been created in Java as the result of string concatenations as if it were prepared statements with parameter markers.
- Together with IBM Optim™ Development Studio it allows for some impact analysis. For example, what happens when the name of a column in table xyz is changed or which SQLs run against column abc.

To use pureQuery in an IMS environment, install either of the following:

- IBM Data Studio: An integrated, modular environment for database development. It can be downloaded at no charge from the following site:

  http://www-142.ibm.com/software/products/us/en/data-studio
- Optim Development Studio: An integrated database development environment that is now included in Data Studio.

Both of these products can be installed stand-alone or in an existing Eclipse/Rational Developer for System z/Rational Application Developer. The client makes the decision on what to use.

Details on pureQuery, its configuration, and how to make it work are discussed in *Using Integrated Data Management To Meet Service Level Objectives,* SG24-7769.

2.6 JVM tuning considerations

The tuning in the JVM environment has to be carefully done with regard to the planned workload. The storage tuning is especially important because the JVM is a memory-managed environment.

An untuned JVM has impact on memory usage, can significantly increase the CPU usage, and can result in higher costs. As soon as the speciality engines, such as System z Application Assist Processor (zAAP) or System z Integrated Information Processor (zIIP), do not have enough capacity, the workload floats to the expensive general-purpose processors. There have been client cases that resulted in critical situations because they went into production without tuning the JVM environment.

JVM tuning in this context consists of carefully adjusting the IBM Language Environment® storage parameters plus the adjustment of JVM storage parameters. The Language Environment runtime options for the storage report and diagnostic tools such as IBM Health Center can provide a good indicator if the storage options were chosen well.

2.7 Debugging Java applications in IMS

Rational Developer for System z, and Rational Application Developer, have an integrated Java debugging perspective that allows a developer to connect to a remote JVM and do interactive debugging of the application.

The IMS Batch Terminal Simulator tool also supports IMS Java batch processing applications (JBPs). Therefore, the debugging of DL/I calls is possible with the same tooling that has existed for traditional IMS batch applications. The new version of IMS Batch Terminal Simulator has a new Eclipse-based GUI, which allows for more integration, for example when the Java code is developed by using an Eclipse-based GUI.

For further details about IBM IMS Batch Terminal Simulator for z/OS, see the following site:

http://www-01.ibm.com/software/data/db2imstools/imstools/imsbts

2.7.1 Debugging a Java BMP region

In this section, we provide an example about how to debug an IMS Java batch application running on the mainframe. The debugging of an IMS online application is similar.

To enable debugging, two additional Java configuration switches are required for JVM:

- Turn on the debug option and tell the JVM to be the server for the debug session
- Suspend execution of JVM until after a debug connection socket from the client is attached

Use the following steps to debug a Java batch message processing (BMP) region:

1. Issue the **-Xdebug** option to tell JVM to turn on debug mode and then issue the **-X runjdwp** option with the parameters: **server=y**, **suspend=y**, **address=7778**, as shown in Example 2-1.

Example 2-1 JVM option for JVM wait

```
-Xdebug
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=7778
```

These parameters tell the JVM that there is a server waiting for an incoming socket connection on port 7778 for a debugging session. It should suspend any work until the debugging session with the Eclipse client (can also be Rational Application Developer or Rational Developer for System z for Java debugging) is established. This means that without a connection, the JVM will not do any work. Depending on the Eclipse version, it is possible that the JVM is initially suspended and will not start executing until resume has been clicked.

2. After the configuration change, start the IMS BMP. It produces the following output:

```
Listening for transport dt_socket at address: 7778
```

   We used a PSB that points to a program called *SimpleRuleEngineRunner*. Then, it is required to select the Java class source in the Rational Developer for System z workspace.

3. Select **Debug As → Debug Configurations**. The Debug Configurations window opens, as shown in Figure 2-1.

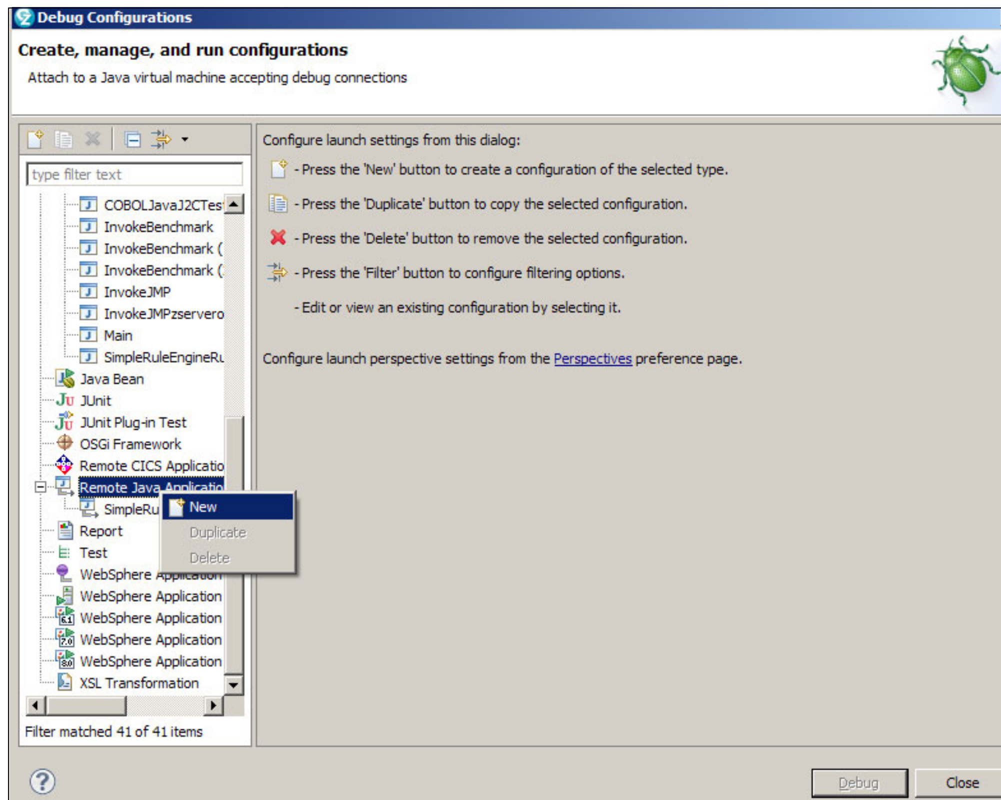


Figure 2-1 Debug configurations window

4. Complete the server host name or IP address and the port number. Select **Socket Attach** as the connection type and check **Allow termination of the remote VM** to avoid hanging IMS BMPs at the end of program execution or if the debug session socket breaks.

   Click **Apply** to save the values and then click **Debug** to connect to the waiting JVM on z/OS. See Figure 2-2 on page 14.

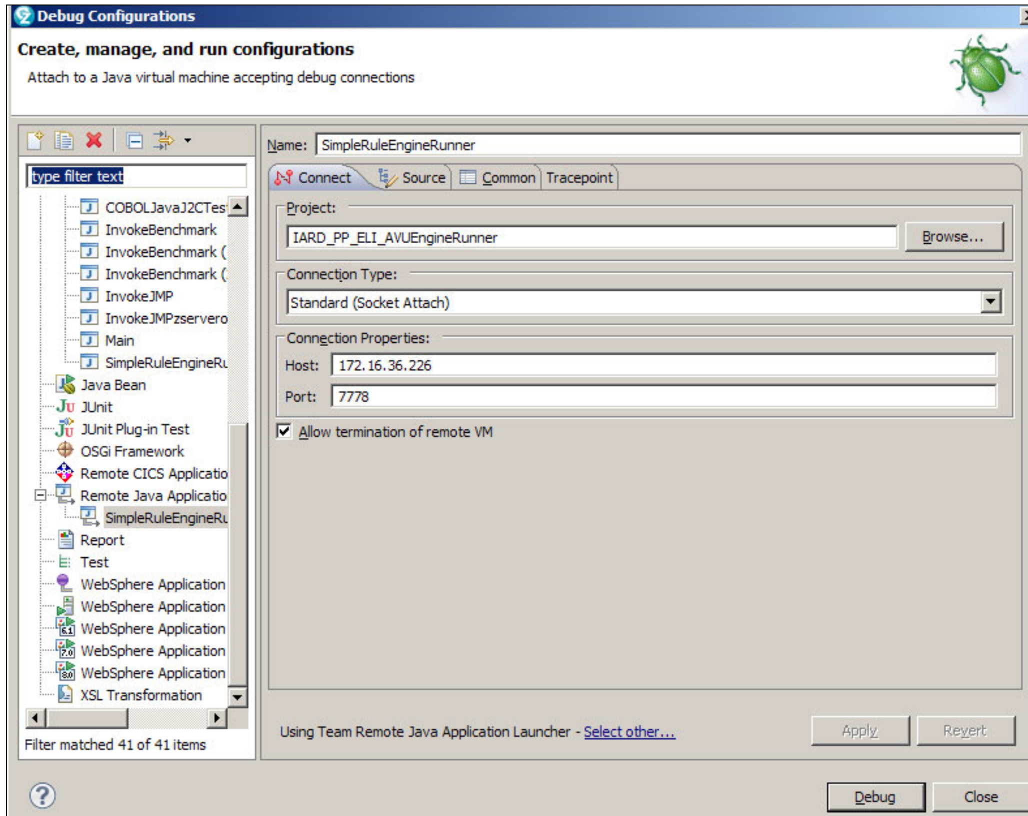


Figure 2-2 Remote Java application properties

> **Note:** If there is no automatic switch to the debug perspective within Rational Developer for System z, switch the perspective manually. The debug session shows up, as shown in Figure 2-3.

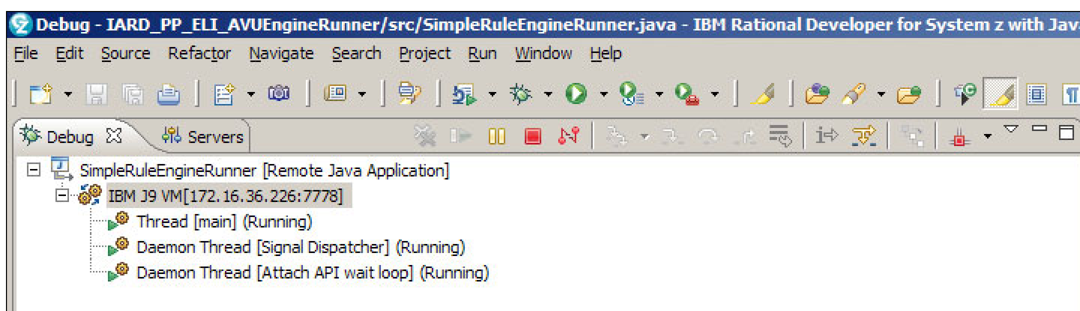


Figure 2-3 Remote debug connection

5. Initially, because there is no breakpoint specified, the Java program starts running as soon as the debug client has connected to the JVM. If the network connection is slow enough or the suspend icon is clicked fast enough, the execution will be suspended and it can be determined where in the source code the execution has been suspended. See the list of suspended tasks in Figure 2-4 on page 15.

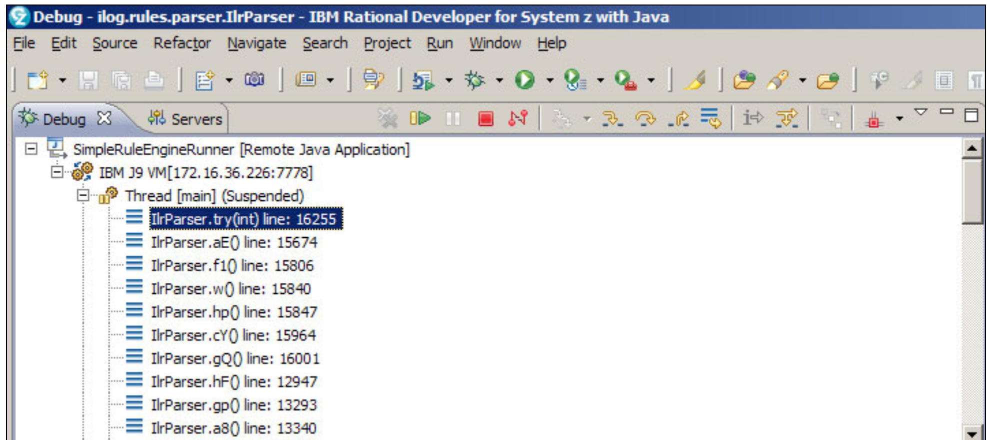


Figure 2-4   Suspended remote JVM execution

6. Scroll down to the most upper level class. It can be seen where in the program source code the execution has been suspended. It also displays the current state/values of variables/objects as shown in Figure 2-5.

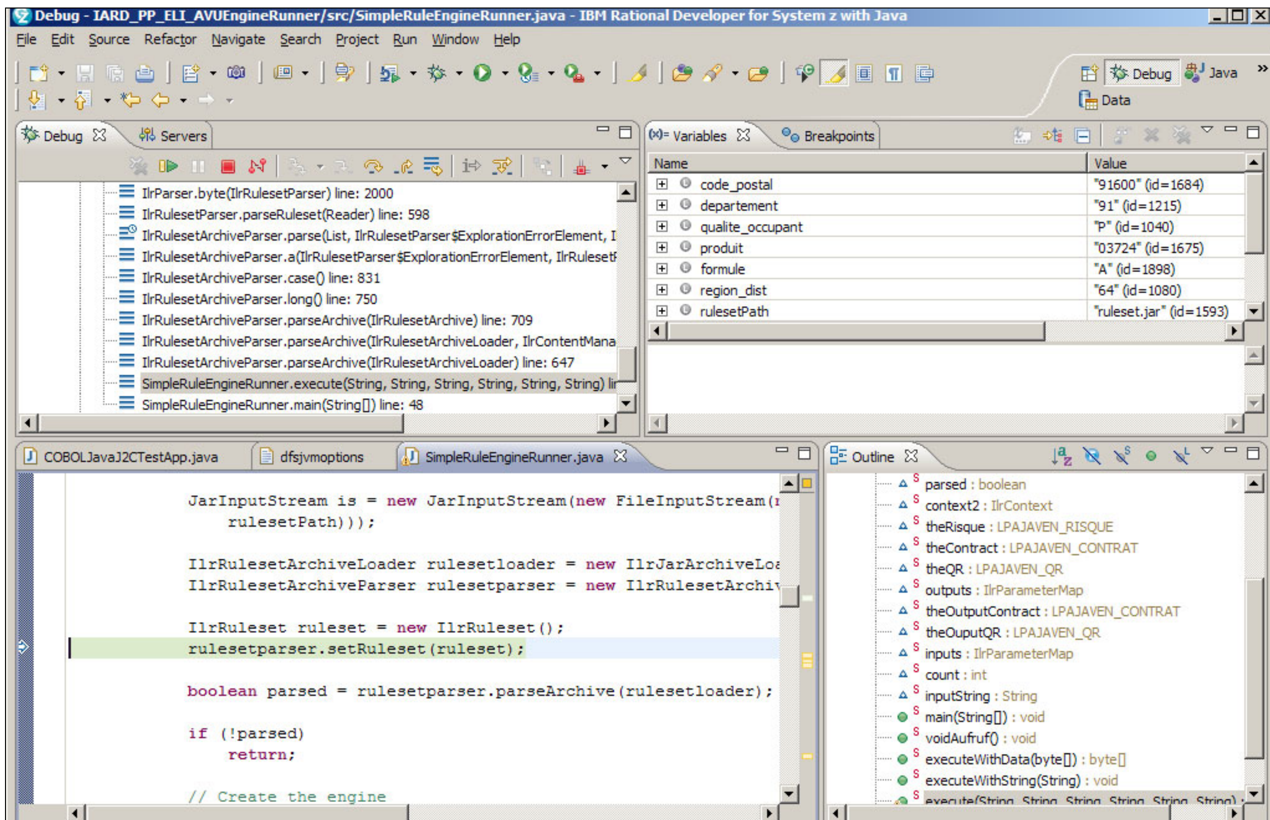


Figure 2-5   Suspended remote Java program with source code position and variable/object values

7. When the execution is suspended or stopped at a breakpoint, the values of objects and variables can also be seen by moving the mouse over variables/objects. See Figure 2-6 on page 16.

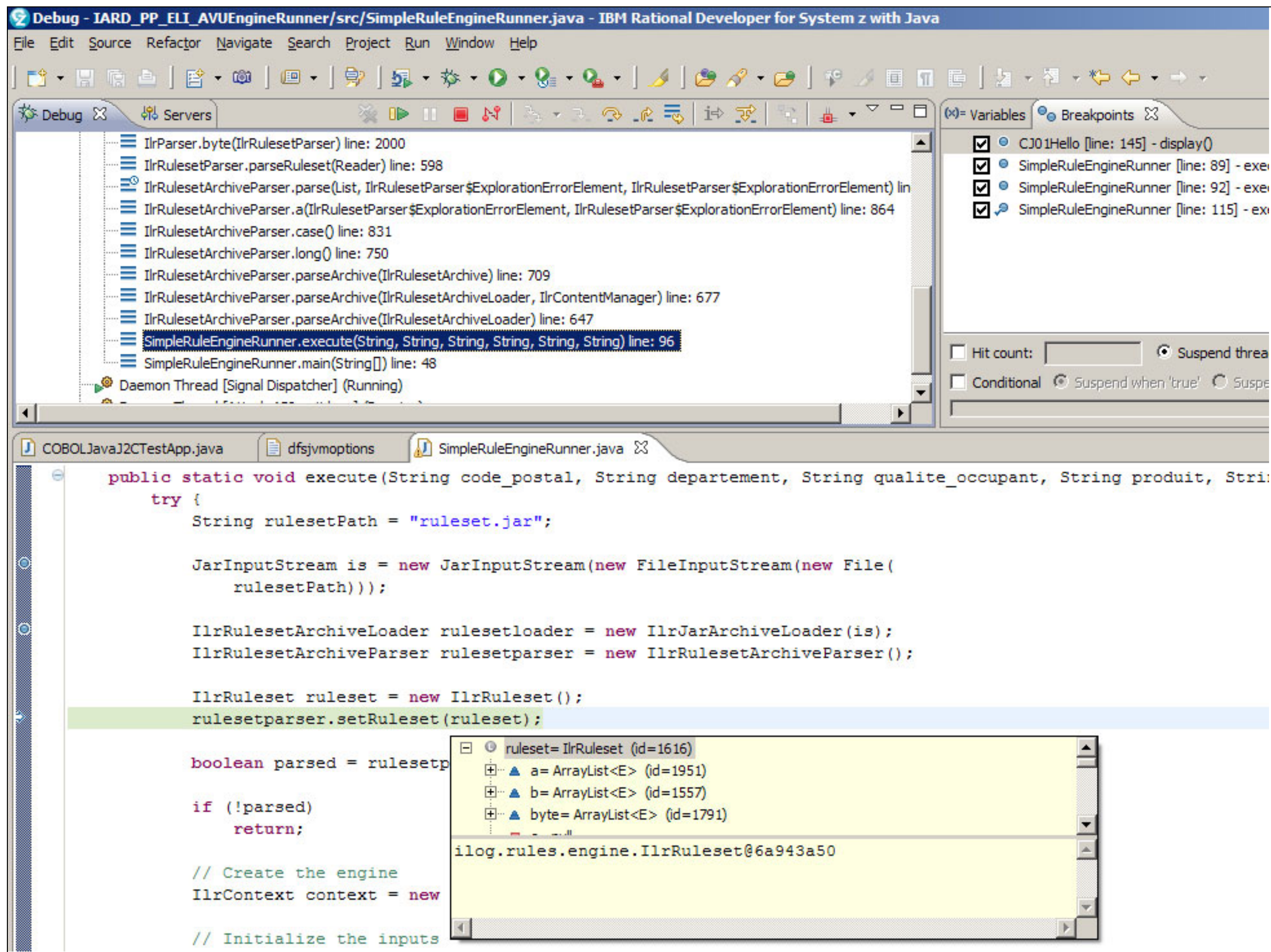


Figure 2-6 Context-sensitive source code view of current variable/object values at suspend/breakpoint time

8. When breakpoints are set and reached, the execution stops at the breakpoint and the line of code is displayed in the source view, as shown in Figure 2-7 on page 17.

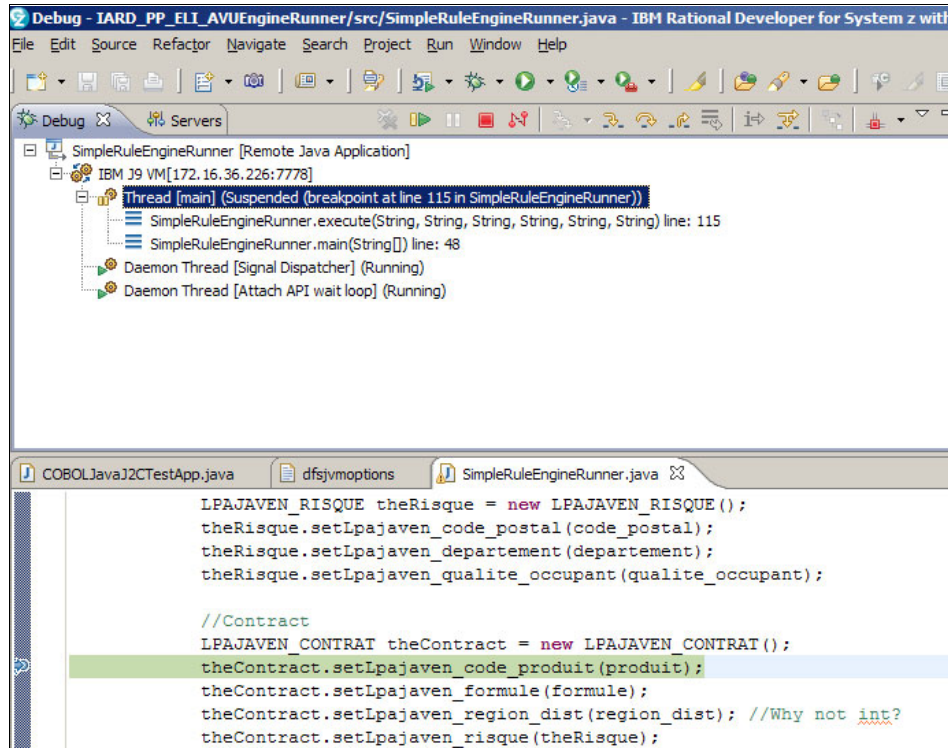


Figure 2-7 Debug session suspended at breakpoint with source code view

9. When the step-through ends, the JVM and the BMP on z/OS terminates and the debug session ends.

2.8 Diagnostics and monitoring of Java in an IMS environment

The IBM Monitoring and Diagnostic Tools for Java - Health Center (Health Center) or Java Monitoring and Management Console (JConsole) can be used for monitoring and diagnostics for Java applications running in IMS. IBM HeapAnalyzer can be used to analyze heap dumps.

The standard z/OS JDK is used for Java in IMS. All tools that work with a standard JDK should also work with IMS.

2.8.1 IBM Monitoring and Diagnostic Tools for Java - Health Center

The Health Center can be used as a diagnostic tool for applications running in the JVM. You can also use the Health Center to monitor the JVM. See the following site:

http://www.ibm.com/developerworks/java/jdk/tools/healthcenter

The configuration is simple. A port is defined in the JVM configuration, which is then used by the Windows based tool to connect to the JVM.

One drawback is the need for a different port for every IMS region. There is no management console or variable support to achieve this, so the best way is for every IMS Java region to have a separate configuration member that has a specific port number hardcoded:

- The z/OS JVM parameter **`-Xhealthcenter`** is used to configure the port

- ► Select a port of choice and add this parameter to the Java configuration: `-Xhealthcenter:port=1982`
- ► The IMS region job log should then print this message: `UTE115: Trace buffer discarded.` The count of discarded buffers is printed at VM shutdown

At the time of writing this book, the Health Center port online became active after calling the first Java method.

To connect from the Health Center to the port, the following steps are required:

1. Install IBM Support Assistant Workbench
2. Install and enable IBM Health Center plug-in
3. Start IBM Support Assistant (ISA)
4. Launch activity: Analyze problem
5. Select IBM Health Center
6. Connect to the JVM of a specific IMS region

The first two steps can be skipped when the Health Center is already installed. The Health Center profiling view as part of IBM Support Assistant is shown in Figure 2-8.

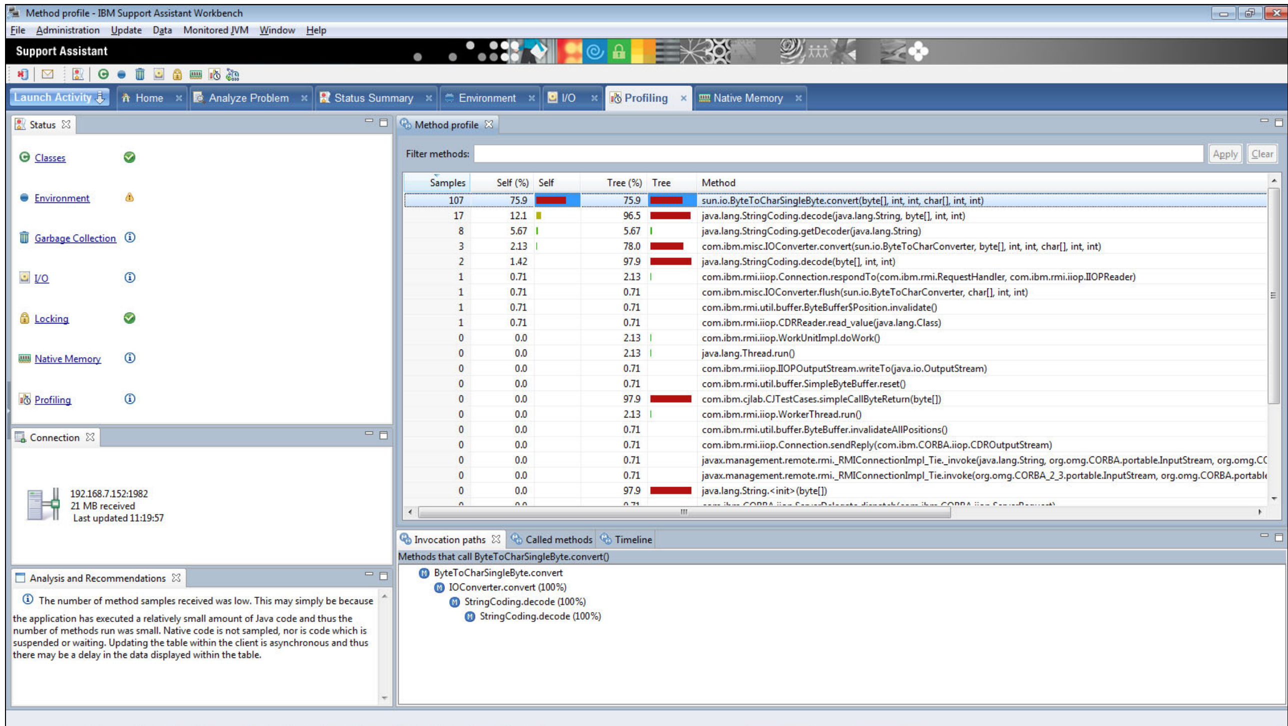


Figure 2-8 Health Center profiling view

> **Note:** The Health Center is not a monitoring tool. It can only connect to and analyze data from one JVM or one IMS region. There is no aggregation functionality available.

2.8.2 Rational Agent Controller, Rational profiling, and Healthcenter Plug-in

Starting with Rational Application Developer V9, both trace-based profiling and sample-based profiling are available. Both require the activation of JVMTI-based libraries on z/OS plus using the Rational Agent Controller address space on z/OS. The benefit is that not every IMS region requires its own port (as with using the **`-Xhealthcenter`** parameter). Instead Rational Agent

Controller only has one port and allows you to select the JVMs based on the process ID. The process ID of the JVM is printed in the IMS job log after the program is started).

For information about how to download and install the Rational V9 based program, see the following site:

http://www-01.ibm.com/support/docview.wss?uid=swg24035191

There is also an IBM developerWorks® article about how to profile Java applications using Rational Application Developer. See the following PDF:

http://www.ibm.com/developerworks/rational/tutorials/profilingjavaapplicationsusingrad/profilingjavaapplicationsusingrad-pdf.pdf

It is required to start the Java class in the IMS region by running the program as either a batch or an online program. It is not possible to start the program or the Java class execution from the Rational Application Developer wizards.

The profiling is also available from Rational Developer for System z, but it requires the license and the installation version, which includes Rational Application Developer.

2.8.3 JConsole

Another way to perform diagnostics is to use *JConsole.* It is a tool (no charge) that is available with JDK and provides the functionality to allow JConsole to connect to it.

JConsole is a utility that is part of the standard JDKs on distributed platforms. However, the server part that delivers the information is also implemented in the JDK for z/OS.

Use the following steps to start and run JConsole:

1. JConsole can be activated by adding the JVM options to the JVM configuration in the IMS region, as shown in Example 2-2.

Example 2-2 JVM options required to activate JConsole

```
-Djavax.management.builder.initial=
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.port=1099
```

The port will not open until the JVM is fully initialized.

> **Note:** The JVM in an IMS region is not active until after the first call to a Java method. Therefore, it might be required to first run a program that calls a Java method or a Java program if running in a Java dependent region.

2. When JVM is active, switch to the workstation, look for the Java home directory and go to `<JAVA_HOME>/bin`. In that directory, locate an executable file called `jconsole.exe` or `jconsole` for UNIX platforms. It can be started by double-clicking it or it can be started from the command line.

   After it is started, a window opens for a New Connection.

3. You need to enter the host name or IP address and the port number that was defined in the IMS regions JVM properties. The port should not be shared between multiple IMS

regions. Each region requires its own port. User name and password are not required here because the security has been disabled with the configuration settings mentioned earlier.

4. After the connect has finished, the GUI opens. You can look at certain things in the JVM, such as heap, threads, and so on.

   Initially, the Overview display opens.

For more information about what diagnostics are possible with JConsole, refer to the documentation at the following site:

http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/index.jsp?topic=%2Fcom.ibm.java.aix.70.doc%2Fdiag%2Ftools%2FJConsole.html

> **Restriction:** Not all functions that the JConsole provides can be used while connected to a JVM that was written by IBM and runs on z/OS.

2.8.4 IBM HeapAnalyzer

You can use *IBM HeapAnalyzer* to analyze Heap memory leaks when running Java applications. Possible failures in that situation could be either S0D4 abends or java.lang.OutOfMemoryError exceptions.

IBM HeapAnalyzer is described and can be downloaded from the following website:

https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=4544bafe-c7a2-455f-9d43-eb866ea60091

A Heap dump in an IMS JVM-enabled environment can be created with the standard procedures for creating heapdumps in UNIX or z/OS environments as described in several IBM documents and post on IBM web pages.

To do an analysis, it is required to create a Heap Dump of the JVM while running the workload that is causing the problem.

An example of the steps to create a heap dump in an IMS JVM-enabled environment is as follows:

1. Set the `-Xdump` option as shown in Example 2-3.

Example 2-3 Option to let create a heap dump in phd format by a user event

```
-Xdump:heap:events=user,opts=phd
```

2. In the environment member add the option shown in Example 2-4.

Example 2-4 Environment variable to be put into DFSJVMEV to enable Java Heap Dumps

```
IBM_HEAPDUMP=true
```

3. Use the **`kill -3`** or **`kill -QUIT`** command followed by the process ID of the JVM and a heap dump can be produced.

   The PTF for APAR PM50971 prints the process ID (PID) for the JVM in the IMS job log, as shown in the sample in Example 2-5.

Example 2-5 Sample output from the IMS job log printing the JVM PID

```
DFSJVM00: -Xmaxf0.8
DFSJVM00: -Xminf0.3
DFSJVM00: -Xmx64M
```

```
DFSJVM00: -Xmso512k
DFSJVM00: -Xss256k
DFSJVM00: -Xms32M
DFSJVM00: -Xcodecache10M
DFSJVM00: -Xshareclasses:name=cobolims1
DFSJVM00: -Xscmx64M
DFSJVM00: -Xscminaot16M
DFSJVM00: +++++++++++++++++++++++++++++++++++++++++
DFSJVM00: ++   End   Contents of -Xoptionsfile   ++
DFSJVM00: +++++++++++++++++++++++++++++++++++++++++
DFSJVM00: JVM initialization started:  Fri Apr 20 17:22:59.793 2012
DFSJVM00: JVM initialization complete: Fri Apr 20 17:23:00.118 2012
DFSJVM00: Process ID:::::::: PID =2175
DFSJVM00: Parent Process ID: PPID=1
DFSJVM00: Process Group ID:: PGID=2175
```

This makes it easy to send signals to the JVM with the kill command in order to create Heap dumps in the format required for the IBM Heap Analyzer.

4. Issue the **kill -QUIT** or **kill -3** command with the PID from the IMS joblog. The messages shown in Example 2-6 will display in the IMS job log. It indicates the successful creation of the PHD Heapdump file and other diagnostic information.

Example 2-6 Output indicating the creation of a successful heap dump

```
JVMDUMP039I Processing dump event "user", detail "" at 2012/07/03 15:35:58 -
please wait.
JVMDUMP032I JVM requested System dump using
'GAEBLER.JVM.TDUMP.CJTSTMPP.D120703.T153558' in response to an event
IEATDUMP in progress with options
SDATA=(LPA,GRSQ,LSQA,NUC,PSA,RGN,SQA,SUM,SWA,TRT)
IEATDUMP success for DSN='GAEBLER.JVM.TDUMP.CJTSTMPP.D120703.T153558'
JVMDUMP010I System dump written to GAEBLER.JVM.TDUMP.CJTSTMPP.D120703.T153558
JVMDUMP032I JVM requested Heap dump using
'/u/gaebler/heapdump.20120703.153558..50334771.0003.phd' in response to an
event
JVMDUMP010I Heap dump written to
/u/gaebler/heapdump.20120703.153558.50334771.0003.phd
JVMDUMP032I JVM requested Java dump using
'/u/gaebler/javacore.20120703.153558.50334771.0004.txt' in response to an
event
JVMDUMP010I Java dump written to
/u/gaebler/javacore.20120703.153558.50334771.0004.txt' in response to an
JVMDUMP013I Processed dump event "user", detail "".
```

5. The `.phd` file can now be downloaded in binary format and loaded into the IBM Heap Analyzer.
6. Start the IBM Support Assistant, and click **Launch Activity** → **Analyze Problem**, select Heap Analyzer and click **Launch**.
7. Select a heap dump file in the local file system by clicking **Browse** and choose the file under the Remote Artifact Browse tab as shown in Figure 2-9 on page 22.

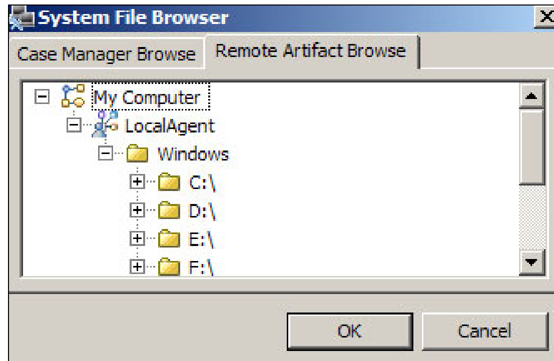


Figure 2-9 Select the Heap Dump phd file location

8. Select the heap dump file from the directory where the binary download of the `.phd` file was stored, as shown in Figure 2-10.

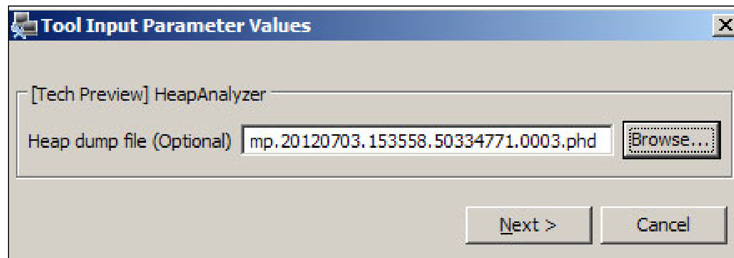


Figure 2-10 Select the phd file

9. Click **Next** and the analysis starts. The IBM HeapAnalyzer starts and displays a summary view of the data gathered, as shown in Figure 2-11 on page 23.

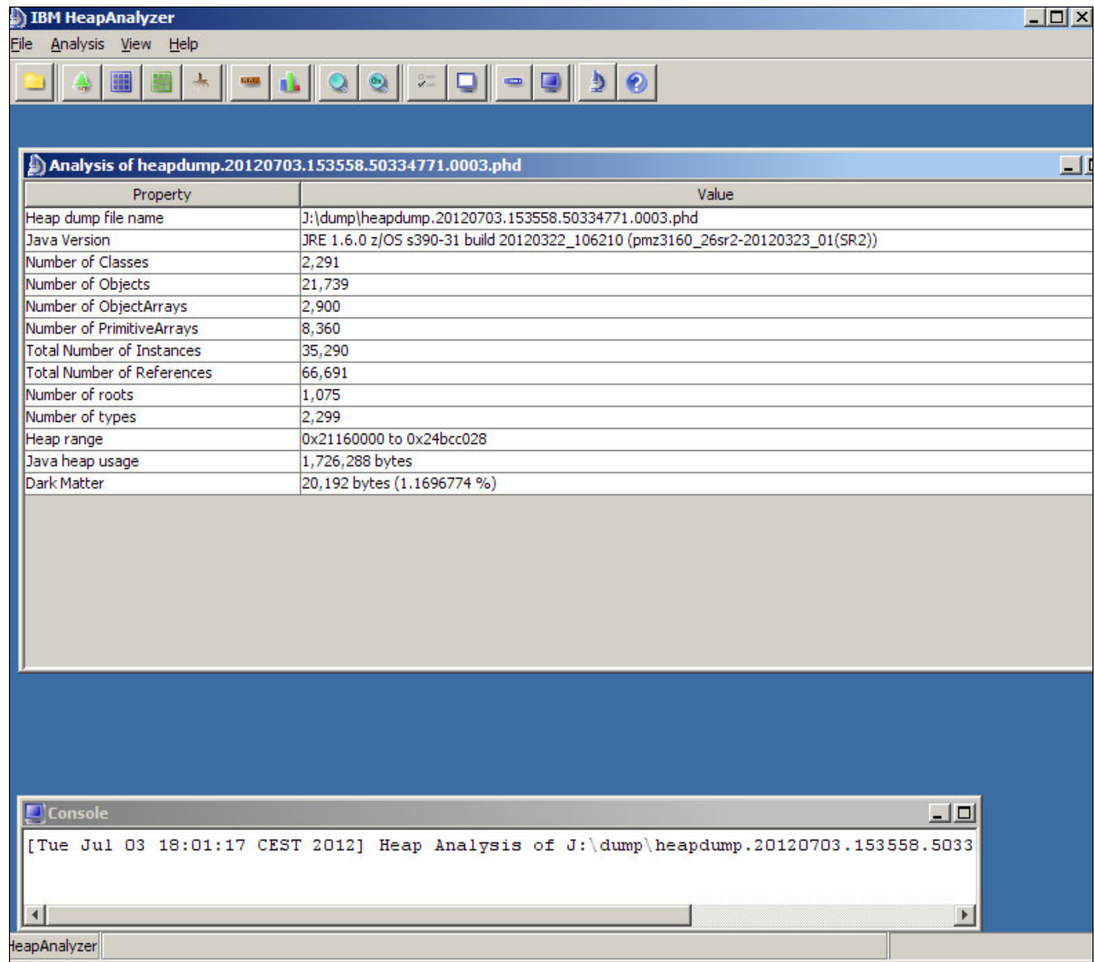


Figure 2-11 Summary view of IBM Support Assistants Heap Analyzer

This example shows how a JVM running attached to IMS can be analyzed with standard tooling just like any other JVM.

2.8.5 Profiling applications

For profiling Java applications or the Java part of an application that run in an IMS region, use the profiling functionality of IBM Health Center or Rational Agent Controller based profiling, which are described in 2.8, “Diagnostics and monitoring of Java in an IMS environment” on page 17. This also applies to message processing programs (MPPs) running a JVM.

The profiling of a Java application can also be done as part of the development cycle. When the Java code has no dependencies for running in IMS, the development environments profiling tools can be used. For example, Eclipse has an integrated profiling infrastructure called *JVM monitor*.

2.8.6 Monitoring JVMs in IMS regions

Currently, there are no JVM monitoring tools available specifically for IMS environments for the monitoring of the JVM. However, for transaction rate, total region memory usage, and so on, the standard IMS monitoring tools can be used.

For JVM specifics such as garbage collection, heap usage, and total JVM memory monitoring, there are tools currently available on the market that can be used that exploit the JVMs JVMTI interface. If these can connect to multiple JVMs, multiple ports and aggregate data, or provide a dashboard, they should be suitable for this purpose. In addition, ISV software can be checked to determine if it meets the requirements.

2.9 Java interoperability with COBOL in IMS batch applications

In this section, we review COBOL calls to Java and vice versa, and how a new IMS Java batch application can interface with existing COBOL modules. The description and samples we have included are IMS based but can be used with minimal changes in the configuration outside IMS because language interoperability is a z/OS Language Environment functionality.

While the JVM in a batch application always persists over the lifetime of the application region, it is not required for batch. This allows the configuration of the JVM parameters with IMS PROCLIB configuration members, which are easier to use and to manage, instead of pointing to the HFS files as was required for the old implementation.

2.9.1 How Java can call COBOL and vice versa

In this section, we describe and provide an example on how a batch application can use COBOL Java interoperability. Most of the statements also apply to other languages, such as PL/I or Assembler, but the INVOKE syntax to call Java methods is unique to COBOL. Java has the option to call and to be called from other languages by providing APIs and an interface called *Java Native Interface* (JNI). JNI was developed for the C/C++ languages, but on z/OS it is available for all Language Environment compliant languages.

Java calls to COBOL and vice versa are the two possible execution environments.

Java calls COBOL

A Java main method starts the application and calls COBOL. Cascading calls Java-COBOL-Java-COBOL are possible. The runtime environment is an IMS Java region for example, Java Batch Program (JBP).

- Due to Language Environment requirements, the COBOL code that is called from Java is required to be OO COBOL classes, but has the option of implementing CALL statements to procedural COBOL modules. It can be static or dynamic, whereas dynamic calls can only be made to DLL-compiled COBOL modules, a statically linked Wrapper module for a NODLL module also meets this requirement.
- Due to Language Environment requirements, the COBOL code that calls Java is required to be mixed case, long name support, and DLL compiled COBOL code. If the caller is NODLL compiled, a statically linked wrapper module for a DLL module is required.

  Figure 2-12 on page 25 depicts Java calls to the Language Environment languages through the JNI.

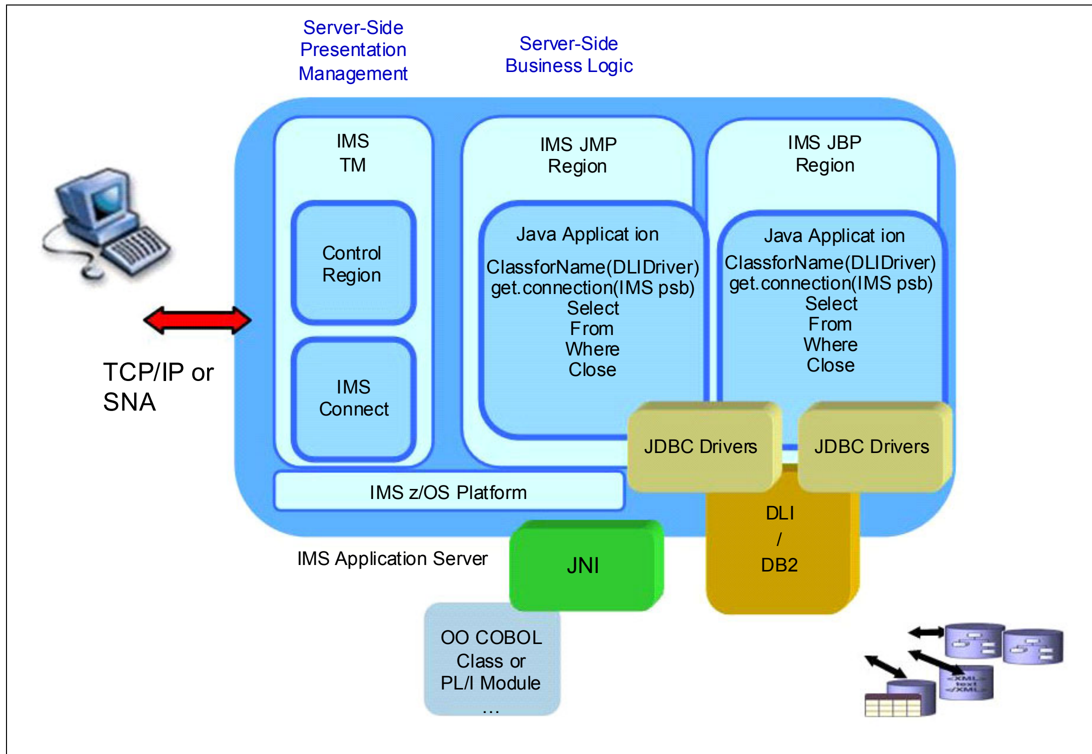


Figure 2-12 Java calls Language Environment languages through the JNI

COBOL calls Java

A COBOL main application starts and calls one or more Java methods. Cascading calls COBOL-Java-COBOL-Java are possible. The runtime environment can be an MPP or BMP.

- Due to Language Environment requirements, the COBOL code that calls Java is required to be mixed case with long name support and DLL compiled COBOL code. If the caller is NODLL compiled, a statically linked Wrapper module for a DLL module is required.
- Due to Language Environment requirements, the COBOL code that is called from Java is required to be OO COBOL classes, but has the option of implementing CALL statements to procedural COBOL modules. It can be static or dynamic, whereas dynamic calls can only be made to DLL compiled COBOL modules. A statically linked Wrapper module for a NODLL module also meets this requirement.

Figure 2-13 on page 26 depicts the Java method of COBOL main calls.

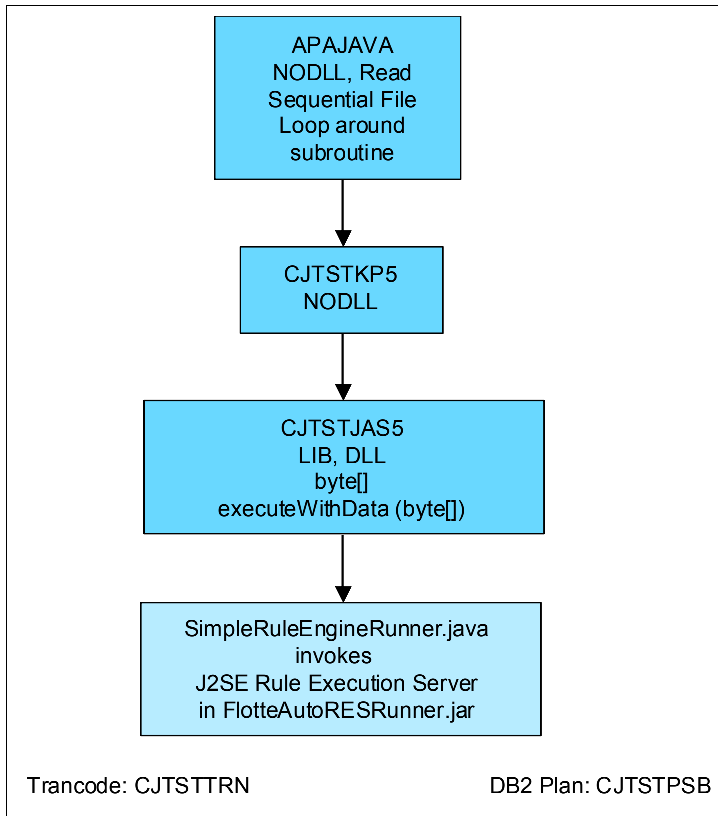


Figure 2-13 COBOL main calling a Java method

This example shows that:

- APAJAVA is traditionally NODLL compiled.
- CJTSTJA5 needs to be DLL compiled in order to call Java.
- To be able to call CJTSTJA5 dynamically from APAJAVA, it needs to have a wrapper.
- DLL modules cannot dynamically call NODLL modules and vice versa.
- But DLL and NODLL modules can be statically linked together.

Solution depicted:

- CJTSTKP5 is NODLL compiled and statically linked together with DLL compiled module CJTSTJA5.
- APAJAVA is NODLL compiled and can then dynamically call CJTSTKP5.

The sample program listed in this figure is also used as basis for the code evolution description in 2.9.3, “COBOL code evolution” on page 27.

These options also apply to other languages, for example, for the Language Environment conform assembler (DLLs) and Enterprise PL/I applications. The difference to COBOL is that the JNI calls and its function pointers are more difficult to implement because there is no built-in support for interfacing with Java. For example, a DLL can also be created with Language Environment Assembler Macros and with PL/I, but the Java wrapper stub source is only generated by the COBOL compiler and must be manually created for the PL/I and Assembler-based DLLs.

The CEEENTRY macro allows it to create Language Environment-compliant DLLs written in assembler. By manually creating a Java class as the JNI wrapper, it is possible to call an

Assembler DLL from Java. For an example, refer to 5.2.2, “Sample application IMS BMP COBOL calls Java” on page 74.

2.9.2 JNI calls using COBOL INVOKE

COBOL has the option to use a simple programming construct to call a Java class using INVOKE. The syntax for the COBOL INVOKE statement and examples can be found in the *Enterprise COBOL for z/OS Programming Guide* at the following web page:

http://pic.dhe.ibm.com/infocenter/pdthelp/v1r1/index.jsp?topic=%2Fcom.ibm.entcobol.doc_4.2%2FPGandLR%2Fref%2Frlpsinvo.htm

Measurements for a client sample showed there is significant overhead with the use of the INVOKE statement when calling Java. Therefore, it is a good practice to implement JNI calls for COBOL. For PL/I the only option is to use JNI calls.

A sample COBOL main application that does loop processing records from a sequential file and calls a COBOL subroutine that then calls a Java method took approximately five days using the COBOL INVOKE statement and 35 seconds of elapsed time with optimized JNI calls. This does not take into account the CPU time used for both cases.

These differences are discussed in more detail in 2.9.3, “COBOL code evolution” on page 27.

2.9.3 COBOL code evolution

In this section, we discuss COBOL code evolution and review the syntax that is used in the enterprise for z/OS COBOL compiler. The samples in this section show the most efficient way of interfacing with Java by using JNI calls from COBOL. For this discussion, it is assumed that there is a subroutine that implements the calls to the Java method and a possible loop around to the calls to Java is implemented in the calling routine.

Basically, the calling routine of the application reads input records from a file, writes the output record, and calls a subroutine to process every record. The subroutine passes the information to an J2SE (or POJO) type of Java class that executes WebSphere IBM Operational Decision Manager rules with the IBM Operational Decision Manager execution engine being executed as part of the Java piece executed inside the IMS batch address space.

We provide some performance numbers based on a client sample.

Use Invoke COBOL Syntax and NewStringPlatform function to convert the data to a Java String

The COBOL Java interoperability samples in the *Enterprise COBOL for z/OS Programming Guide*, SC14-7382, provide a convenient and code-friendly way of calling Java methods.

The COBOL INVOKE statement is used with the NewStringPlatform conversion routine that allows the conversion of a PIC X string into a Java string object (java.lang.String). See Example 2-7.

Example 2-7 COBOL calls Java using Invoke syntax code snippet

```
* Put CHAR we got from caller to helper variable
           Move JAVAIN to stringBuf
      * Convert helper variable to jstring Object instance
           Call "NewStringPlatform"
             using by value JNIEnvPtr
```

```
                     address of stringBuf
                     address of jstring1
                     0
        returning rc
      * Otherwise pass jstring to Hello class using its init method
          Invoke Hello "simpleCall" using by value jstring1
      *        returning jint1
```

The sample workload of 10 billion calls to the subroutine produced the SMF30 accounting summary that is shown in Example 2-8.

Example 2-8 SMF30 accounting for 10 billion calls with COBOL calls Java invoke syntax

```
S t e p   E n d   S t a t i s t i c s

Step Name: G              Cond Code: 0000        Start: 30-Jan-2012 02:11:08 PM
Step  Num: 1              PGM  Name: DFSRRC00    End:   03-Feb-2012 05:50:02 PM
CPU (TCB): 02:34:59.09                           Storage below 16M:       260k
CPU (SRB): 00:00:23.55                           Storage above 16M:    277,368k
CPU (ZIE): 00:00:00.00                           64bit prv Storage:         0k
CPU (ZIP): 00:00:00.00
CPU (IFE): 02:34:59.00
CPU (IFA): 96:48:39.27
CPU (ALL): 99:24:01.91
Trans Act:403:38:54.71                           Service Units:    520,579,178
Tape Mnts: 0                                     Total EXCPs:           25,510

Unit-- DDName-- EXCP Count-- Blksize       Unit-- DDName-- EXCP Count-- Blksize
B021 D STEPLIB            56  32,760       B023 D STEPLIB            38  32,760
B123 D STEPLIB            50  32,760       B023 D DFSESL              4  32,760
B022 D DFSESL              1  32,760       B020 D PROCLIB            16   8,800
B021 D DFSCTL              2   3,200       B121 D SYS00002            2   6,160
B121 D SYS00005            4     256

Total DASD EXCPs:                173       Total Tape EXCPs:                  0
```

The elapsed time is slightly more than five days.

Using JNI calls as in JDK C samples and passing byte arrays

The next iteration of the code is to use JNI functions and pass the string as a byte array to Java and use Java methods to convert the byte array into a Java string object. The call sequence required in COBOL is as follows:

- Use NewByteArray to create a Bytearray
- Use SetByteArrayRegion to move the COBOL structure/PIC X into the ByteArray
- Use Findclass to create a class reference for the Java Class method invocation
- Use GetStaticMethodID to get the MethodID for the next call
- Use CallStaticObjectMethod to invoke the Java method
- Use GetByteArrayRegion to retrieve the results Byte Array from Java
- Use DeleteLocalReference to remove the Input and Output Byte Arrays

The sample workload of 10 billion calls to the subroutine produced the SMF30 accounting summary that is shown in Example 2-9 on page 29.

Example 2-9 SMF30 accounting for 10 billion calls with JNI functions and Byte Array

```
S t e p   E n d   S t a t i s t i c s

Step Name: G             Cond Code: 0000        Start: 25-Jan-2012 06:53:04 PM
Step  Num: 1             PGM  Name: DFSRRC00    End:   26-Jan-2012 04:34:04 AM
CPU (TCB): 06:10:30.84                          Storage below 16M:        260k
CPU (SRB): 00:00:03.66                          Storage above 16M:    422,780k
CPU (ZIE): 00:00:00.00                          64bit prv Storage:          0k
CPU (ZIP): 00:00:00.00
CPU (IFE): 06:10:30.80
CPU (IFA): 03:27:50.40
CPU (ALL): 09:38:24.90
Trans Act: 09:40:59.33                          Service Units:     935,248,976
Tape Mnts: 0                                    Total EXCPs:            17,154

Unit-- DDName-- EXCP Count-- Blksize       Unit-- DDName-- EXCP Count-- Blksize
B021 D STEPLIB            54  32,760       B023 D STEPLIB            32  32,760
B123 D STEPLIB            50  32,760       B023 D DFSESL              4  32,760
B022 D DFSESL              1  32,760       B020 D PROCLIB            16   8,800
B021 D DFSCTL              2   3,200       B121 D SYS00002            2   6,160
B121 D SYS00005            4     256

Total DASD EXCPs:            165       Total Tape EXCPs:                  0
```

The elapsed time (and CPU usage) decreased significantly but was still almost 10 hours.

Put the loop inside the subroutine

Putting the loop inside the subroutine to obtain the number of looping cycles is one of the most efficient ways of performing the calls.

The code was changed to have no JNI overhead as the ideal reference case to call Java. The loop was moved from the calling routine into the subroutine for test purposes. But this might not be possible for all client situations or applications, because sometimes the existing call sequences have to be preserved:

1. The following calls are done once:
   a. Use NewByteArray to create a Bytearray
   b. Use SetByteArrayRegion to move the COBOL structure/PIC X into the ByteArray
   c. Use Findclass to create a class reference for the Java Class method invocation
   d. Use GetStaticMethodID to get the MethodID for the next call
2. The loop was built around the following calls:
   a. Use CallStaticObjectMethod to invoke the Java method
      perform CALLMETHOD THRU CALLMETHOD-END DURCHLAEUFE TIMES
   b. Use GetByteArrayRegion to retrieve the results Byte Array from Java
   c. Use DeleteLocalReference to remove the Output Byte Array
3. After the loop ends cleanup is required:

   Use DeleteLocalReference to remove the Input Byte Array

A sample workload of 10 billion calls to the subroutine produced the SMF30 accounting summary that is shown in Example 2-10 on page 30.

Example 2-10   SMF30 accounting for 10 billion calls with the loop inside the subroutine

```
S t e p   E n d   S t a t i s t i c s

Step Name: G              Cond Code: 0000        Start: 25-Jan-2012 06:25:18 PM
Step  Num: 1              PGM  Name: DFSRRC00    End:   25-Jan-2012 06:25:29 PM
CPU (TCB): 00:00:03.51                           Storage below 16M:        260k
CPU (SRB): 00:00:00.00                           Storage above 16M:    352,320k
CPU (ZIE): 00:00:00.00                           64bit prv Storage:          0k
CPU (ZIP): 00:00:00.00
CPU (IFE): 00:00:03.45
CPU (IFA): 00:00:07.79
CPU (ALL): 00:00:11.30
Trans Act: 00:00:11.43                           Service Units:         642,768
Tape Mnts: 0                                     Total EXCPs:            25,331

Unit-- DDName-- EXCP Count-- Blksize       Unit-- DDName-- EXCP Count-- Blksize
B021 D STEPLIB            56  32,760       B023 D STEPLIB            38  32,760
B123 D STEPLIB            50  32,760       B023 D DFSESL              4  32,760
B022 D DFSESL              1  32,760       B020 D PROCLIB            16   8,800
B021 D DFSCTL              2   3,200       B121 D SYS00002            2   6,160
B121 D SYS00005            4     256

Total DASD EXCPs:                173       Total Tape EXCPs:                  0
```

No test case was found where the elapsed time could have been sped up more and where the CPU usage could have been reduced any further. The 11 seconds was the best result when testing this given workload.

Use object caching and WORKING STORAGE with loop outside the subroutine

The sample in this section reviews what needs to be done when designing a solution to call a Java method through a subroutine:

- The sample in "Put the loop inside the subroutine" on page 29 is changed to put the items for JNI one time setup into WORKING STORAGE. This persists for the lifetime of the COBOL module in the Language Environment enclave storage.
- The input byte array reference was created as a global Java object reference, put into WORKING STORAGE, and reused for all inputs.
- All other data items were put into LOCAL STORAGE.
- The loop was moved back to the calling routine. Byte arrays were still passed to Java and Java methods were used to do the conversion to String objects.
- The Working-storage section is used to save the Input Byte Array and Java object reference plus the STATIC-METHOD-ID, see Example 2-11.

Example 2-11   Definition in Working-storage section to limit the JNI overhead

```
Data division.
      Working-storage section.
      01 cached-class-reference object reference Hello value null.
      01 STATIC-METHOD-ID       PIC S9(9) BINARY VALUE 0.
      01 cachedDataByteArray object reference jbyteArray value null.
```

Make sure that the items in the working-storage section are initialized. Otherwise, infrequent 0C4 abends can occur when driving the workload.

Even though the loop is outside the subroutine, the following calls are executed only once:

- Use NewByteArray to create a Bytearray
- Use Findclass to create a class reference for the Java Class method invocation
- Use GetStaticMethodID to get the MethodID for the next call

  The StaticMethodID is always the same for the lifetime of the JVM and it can be reused when the JVM persists during the COBOL subroutine calls.

This sample performs best with the loop in the main COBOL routine. It needs approximately two times more CPU than the loop inside subroutine reference sample, which is unbeaten in terms of elapsed time and CPU usage.

> **Note:** The objects in working storage are never garbage collected for reuse optimization.

A sample workload of 10 billion calls to the subroutine produced the SMF30 accounting summary that is shown in Example 2-12.

Example 2-12 SMF30 accounting for 10 billion calls

```
S t e p   E n d   S t a t i s t i c s

Step Name: G              Cond Code: 0000       Start: 25-Jan-2012 06:35:05 PM
Step  Num: 1              PGM  Name: DFSRRC00    End:   25-Jan-2012 06:35:28 PM
CPU (TCB): 00:00:07.24                          Storage below 16M:        260k
CPU (SRB): 00:00:00.00                          Storage above 16M:     444,600k
CPU (ZIE): 00:00:00.00                           64bit prv Storage:         0k
CPU (ZIP): 00:00:00.00
CPU (IFE): 00:00:07.18
CPU (IFA): 00:00:15.57
CPU (ALL): 00:00:22.81
Trans Act: 00:00:22.93                          Service Units:       1,284,776
Tape Mnts: 0                                    Total EXCPs:            25,380

Unit-- DDName-- EXCP Count-- Blksize        Unit-- DDName-- EXCP Count-- Blksize
B021 D STEPLIB           54  32,760         B023 D STEPLIB           32  32,760
B123 D STEPLIB           50  32,760         B023 D DFSESL             4  32,760
B022 D DFSESL             1  32,760         B020 D PROCLIB           16   8,800
B021 D DFSCTL             2   3,200         B121 D SYS00002           2   6,160
B121 D SYS00005           4     256

Total DASD EXCPs:               165         Total Tape EXCPs:                0
```

The elapsed time is 23 seconds, which is slightly more than two times the best optimized version of code driving the workload. A COBOL coding sample for an optimized JNI sample with an outer loop to call a Java method is shown in Example 2-13 on page 32.

Example 2-13 COBOL coding sample for optimized JNI sample

```
cbl lib,pgmname(longmixed),noexp
       Identification Division.
       Program-id. "CJTSTJA5" recursive.
       Environment Division.
       Configuration section.
       Repository.
           Class Base    is "java.lang.Object"
           Class Integer is "java.lang.Integer"
           Class Hello   is "SimpleRuleEngineRunner"
           Class JavaException  is "java.lang.Exception"
           Class jstring        is "jstring"
           Class jbytearray     is "jbytearray"
           Class jint           is "jint".
       Data division.
       Working-storage section.
       01 cached-class-reference object reference Hello value null.
       01 STATIC-METHOD-ID        PIC S9(9) BINARY VALUE 0.
       01 cachedDataByteArray object reference jbyteArray value null.
       Local-storage section.
       01 ex               object reference JavaException.
       01 DataByteArray object reference jbyteArray value null.
       01 OutputByteArray object reference jbyteArray value null.
       01 class-reference object reference Hello value null.
       01 rc               pic s9(9) comp-5.
       01 laenge           pic s9(9) comp-5 value 50.
       01 offset           pic s9(9) comp-5 value 0.
       01 len              pic 9(9) binary.
       01 testjava.
          02 javainin PIC X occurs 50.
       01 method-name      pic x(50).
       01 class-name      pic x(50).
       01 method-parm-description pic x(50).
       Linkage section.
       01  JAVARC  PICTURE S9(3) COMP.
       01  JAVAIN PIC X(50).
       01  JAVAOUT PIC X(50).
      * Copybook for JNI copybooks and function pointer definitions
       Copy JNI.
       Procedure Division USING JAVARC, JAVAIN, JAVAOUT.
      * Required JNI setup
           Set address of JNIenv to JNIEnvPtr
           Set address of JNINativeInterface to JNIenv
      * Set up the Input Byte Array when not there, create it
           If cachedDataByteArray = null
             Call NewByteArray
               using by value JNIEnvPtr
               by value laenge
               returning DataByteArray
             Call NewGlobalRef
               using by value JNIEnvPtr
               by value DataByteArray
               returning cachedDataByteArray
           Else
             Call NewLocalRef
```

```
          using by value JNIEnvPtr
          by value cachedDataByteArray
          returning DataByteArray
      End-if
      move javain to testjava
      Call SetByteArrayRegion
      using by value JNIEnvPtr, DataByteArray, offset,
        laenge, address of testjava
* Findclass only if class reference not in working storage
      If cached-class-reference = null
        Move 0 to STATIC-METHOD-ID
        Move z"SimpleRuleEngineRunner" to class-name
        Call "__etoa" using by value address of class-name
          returning len
        Call FindClass using by value JNIEnvPtr
          address of class-name returning class-reference
        If class-reference = null
           Display "Error occurred locating TestClass class"
           Goback
        End-if
        Call NewGlobalRef
          using by value JNIEnvPtr
          by value class-reference
          returning cached-class-reference
      Else
        Call NewLocalRef
          using by value JNIEnvPtr
          by value cached-class-reference
          returning class-reference
      End-if
* Get the static method id if not in working storage
      If STATIC-METHOD-ID = 0
        Move z"executeWithData" to method-name
        Call "__etoa" using by value address of method-name
          returning len
        Move z"([B)[B" to method-parm-description
        Call "__etoa" using by value address of
          method-parm-description
          returning len
        call GetStaticMethodID using
                by value JNIEnvPtr
                by value class-reference
                address of method-name
                address of method-parm-description
                returning STATIC-METHOD-ID
        If STATIC-METHOD-ID = 0
           Display "Error occurred while getting STATIC-METHOD-ID"
           Stop run
        End-if
      End-if
* Execute the java method by passing and receiving a byte array
      call CallStaticObjectMethod using
           by value JNIEnvPtr
           by value class-reference
           by value STATIC-METHOD-ID
```

```
          by value DataByteArray
          returning OutputByteArray
* Check for Error
      Perform JavaExceptionCheck
* Get Data Back
      Call GetByteArrayRegion
      using by value JNIEnvPtr, OutputByteArray, offset,
        laenge, address of testjava
      Move testjava to javaout
* Delete Object Reference of the output byte array
      Call DeleteGlobalRef
        using by value JNIEnvPtr
              by value OutputByteArray
* Delete Weak Object Reference of the output byte array
      Call DeleteGlobalRef
        using by value JNIEnvPtr
              by value OutputByteArray
* We're done and return to caller
      Goback
      .
* Check for an Exception occured in Java
   JavaExceptionCheck.
* ExceptionOccured JNI function returns Exception object
      Call ExceptionOccurred using by value JNIEnvPtr
        returning ex
* If its not null then something went wrong
* If you don't catch the Exceptions in the Java code you will
* land here and to go back to Java you need to call another
* method, so better handle all Exceptions in Java try catch
* blocks
      If ex not = null then
* Clear the Exception
        Call ExceptionClear using by value JNIEnvPtr
* Display a message to the Job output
        Display "Caught an unexpected exception"
* Print the Exception Stack Trace to get a clue what went wrong
*       Invoke ex "PrintStackTrace"
* Set error code to 100
        MOVE 100 TO JAVARC
* Return to caller
        MOVE 0 TO RETURN-CODE
        Goback
      End-if
      .
    End program "CJTSTJA5".
```

2.9.4 Latest COBOL INVOKE versus JNI API measurements

In the latest enhancements to the Enterprise COBOL for z/OS compiler, the performance of COBOL INVOKE has greatly improved. Therefore, the use of COBOL INVOKE does not need to be discouraged. COBOL INVOKE is easier to use in version JNI API programming, giving the developer more control with the JNI calls and the opportunity to save some CPU cycles in comparison to using the COBOL INVOKE syntax.

2.9.5 JNI programming considerations

When programming JNI, there are rules that should be followed, especially to allow objects created in COBOL to be eligible for the JVMs garbage collections. If Java objects are not needed anymore in Java, they need to be marked as being eligible for garbage collection by using the JNI call DeleteLocalReference and, for objects made global, the DeleteGlobalReference needs to be used.

It can become bothersome if there are many Java objects created. In this situation, there are JNI calls that create a buffer for all Java objects. Then, without the need to free all references, the buffer can be deleted and all Java object references are gone. This is done by using the PushLocalFrame JNI call at the beginning of the application/module/IMS transaction and using the PopLocalFrame JNI call at the end of the application/module/IMS transaction.

In general, JNI programming is not simple and requires much knowledge. There are few COBOL-based samples, but it is possible to look up all the C-based samples on the web to get the idea about how the JNI calls should be used.

2.9.6 Options to pass data items between COBOL and Java

It is best to pass byte arrays that represent the byte-compatible data in the COBOL application between COBOL and Java. The reason is that usually not all data can be passed as a string or simple types. In addition, since Java methods can only return one data item, it is not possible to return more than one string, byte array, or simple types.

Therefore, use the JNI functions such a *GetByteArrayRegion* and *SetByteArrayRegion* to transform a COBOL structure into a byte array and vice versa. The called Java method then is defined as accepting a byte array and returning a byte array. The marshalling, transformation from COBOL to Java type and back, is then done when accessing the data using a getter or setter method from within Java. This has been proven successful in several client cases and performance tests as an efficient way of passing data back and forth between Java and traditional Language Environment languages.

2.10 Generating Java classes

There are two ways to generate the Java classes that represent the traditional language data structures: J2C wizards and JZOS Record Generator.

Rational Tooling has a wizard to generate byte-compatible Java classes with getter and setter methods for accessing the data fields out of COBOL, PL/I, and C copybooks and structures.

2.10.1 J2C wizards

The J2C wizards for creating a CICS/IMS Data Binding are part of either Rational Application Developer for Java or Rational Developer for System z with Java. The stand-alone Rational Developer for System z does not have the wizards packaged. The two no-cost Rational Developer for System z licenses that come with any IMS version and are downloadable only as stand-alone versions from the IMS website do not have these wizards packaged.

The following steps review how to start the J2C wizard and generate the Java classes:

- The J2C wizard is started with the menu option: **File** → **New** → **Other**. Then, select **J2C** → **CICS/IMS Java Data Binding**.

- A screen capture of the CICS/IMS Java Data Binding wizard starting with the source Data Import is shown in Figure 2-14.

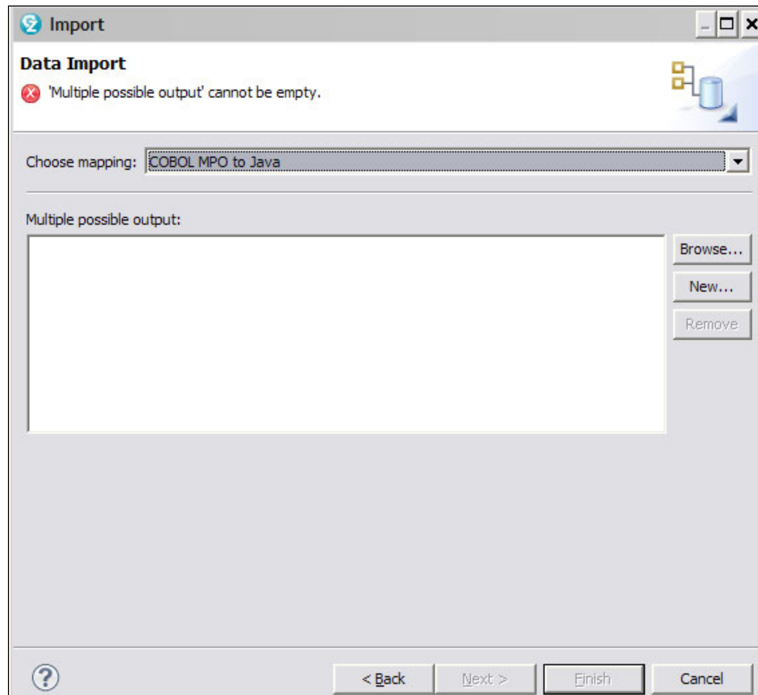


Figure 2-14 Screen capture of the CICS/IMS Java Data Binding wizard

- Go through the wizard to create:
  - CICS/IMS data binding
  - Import, such as the COBOL 01 record type
  - Generate the Java class
- Then, the Java class needs to be populated with the byte array that was passed to the Java method from COBOL. For other languages, such as C or PL/I, the process is identical.

2.10.2 JZOS Record Generator

JZOS Record Generator is the most common way of passing data between COBOL and Java, where copybooks are treated as byte arrays and passed to Java. The unmarshalling into Java fields and data types is done in Java. The JZOS solution can be used from IMS and exploits built-in native JVM functions, which are faster than pure Java data conversions. Although other solutions use the new z/OS batch container, they still do marshalling and unmarshalling when passing data between COBOL and Java.

The JZOS Record Generator uses the ADATA output from the COBOL compiler.

A step-by-step guide for using both JZOS and the J2C approaches is included in the *JZOS for z/OS SDKs Cookbook,* which can be found at the following site:

https://www.ibm.com/services/forms/preLogin.do?source=zossdkcookbook

The JZOS Record Generator also allows conversion of Assembler CSECTs into Java records, and as such, can be quite useful to process data that is defined in the z/OS as Assembler source only. It uses the ADATA output from the Assembler.

2.11 Restrictions for COBOL Java interoperability

There are some known restrictions for COBOL Java interoperability. One is that it requires the IBM Language Environment setting XPLINK(ON). But AMODE24 and VS COBOL modules will not work with XPLINK(ON). They result in a runtime exception including a message.

An example of the message for a call to VS COBOL module with XPLINK(ON) is shown in Example 2-14.

Example 2-14 Error message when trying to run VS COBOL programs with XPLINK(ON)

```
IGZ0186S An attempt was made to run a VS COBOL II program with the run-time option
XPLINK(ON). The program name is...
```

At one client, the conversion of the remaining 30 AMODE24 assembler modules to AMODE31 took about two weeks. They did no separate testing, since testing of the AMODE31 assembler routines was part of the tests for COBOL and Java interoperability.

To support languages other than COBOL, such as PL/I or Assembler, it is required to have an Language Environment-compatible version of PL/I or a Language Environment compatible/enabled assembler routine as the caller. IMS has added a new diagnostic message in case there are non Language Environment-compliant routines:

```
DFS650E NON-LE COMPLIANT PROGRAM IN PERSISTENT JVM ENVIRONMENT, NAME=program_name
```

This should help to find the module that is not Language Environment-compliant. It can be very difficult to find occurrences of old modules still used in production, especially if the modules are object code only and the source is not available in the shop. Some modules have been used for decades. An easy way is to enable the JVM in the IMS regions in the test environment and wait for abends.

2.12 Abend and error handling

If JVM is present in an IMS region, by default it registers a number of signal handlers. You need to make sure that all variables or structures are initialized in subroutines that are called multiple times during the lifetime of the IMS region containing the JVM. The Language Environment enclave in JVM enabled environments is persistent and only destroyed at abends or exceptions. At client sites poorly initialized routines have resulted in rare0C4 abends without a trace of what caused them.

If there is a 0C9 (numeric exception) in the COBOL code, by default there is no 0C9 abend. Rather there is a Language Environment abend such as U4038 or an IMS user abend such as a U101. When an error like this occurs, in order to turn that behavior off and produce a 0C9 abend message, the JVM requires a switch that disables the registration of POSIX handlers by the JVM.

This implementation also ensures that IMS abends, such as 0476 and 0711, make it to the user without the Language Environment and JVM handlers catching it. This allows the continued use of the abend and error handler mechanisms that are in place or used without the JVM being present.

> **Important:** To get an abend with the real abend cause, for example 0C9, it is required to turn on the JVMs standard registration of POSIX handlers. This is done by using the `-Xsignal:userConditionHandler=percolate` JVM command-line option.

Tools such as IBM Fault Analyzer can be used to display the root cause of the error, which is not easy for some Language Environment abends.

Some clients have registered custom Language Environment abend handlers, and if the use in mixed mode environments needs to be continued, turning off the POSIX handler registration by the JVM is preferred. A recent IMS PTF that handles these types of errors was changed to better suit the needs of a production environment. Therefore, make sure to have the latest IMS maintenance applied.

Refer to the manuals for IBM 31-bit SDK for z/OS, Java Technology Edition, V6.0.1, and the SDK Guide at this link:

http://www-03.ibm.com/systems/z/os/zos/tools/java/products/sdk601_31.html#j6content

2.12.1 Alternate options

What if Language Environment Java interoperability is not considered, or is not the preferred option, for mixing existing applications with Java code? A list of possible implementations is discussed in 4.1, "IMS callout to external services" on page 52. Some message-based options, such as IMS callout or WebSphere MQ, are not the preferred options for calling Java methods from batch that should run on the mainframe.

A couple of alternatives that use cross address space calls within the same z/OS image are:

- WebSphere z/OS Optimized Local Adapters. For more information, refer to "WebSphere z/OS Optimized Local Adapters" on page 54
- Calling DB2 Java Stored Procedures

2.13 z/OS considerations

In this section, we describe z/OS considerations for JVM-enabled IMS regions. During testing, when clients ran the JVM in production, there were situations where JNI programming errors led to out-of-storage conditions in the JVM-enabled IMS regions. These regions then failed during normal region termination (Memterm) and entered a state where they could not be stopped using IMS or z/OS commands, including FORCE. The main reason for this was that the modules required for doing the region termination could not be loaded into private storage.

A PMR that was opened to address this issue suggested implementing an IEFUSI exit to reserve 512 k storage below 16 M to allow the region termination modules to be loaded. While this increases the chances for the successful region termination, it is not a guarantee for region termination.

The recommendations from the PMR are shown in Example 2-15 on page 39. It is suggested to use as a base the sample IEFUSI exit provided in member IEEUSI in SYS1.SAMPLIB and replace the statements.

Example 2-15 Sample IEFUSI exit

FROM: USING REGION,R07 ADDRESSABILITY FOR REGION DSECT

```
OI REGFLAGS,X'80' SET THAT IEFUSI CONTROLS REGIONS
TM 0(R08),X'80' V=R JOB?
BO EXIT YES USE DEFAULT VALUES
L R10,REGSZREQ GET REQUESTED REGION SIZE
LTR R10,R10 IS IT ZERO
BZ EXIT YES USE DEFAULT VALUES
AL R10,N64K ADD N64K TO REGION SIZE BELOW
ST R10,REGLIMB SET IT AS REGION LIMIT BELOW
MVC REGSIZB,REGSZREQ SET REGION BELOW
MVC REGSIZA,REG32MB SET REGION SIZE ABOVE TO 32MB
```

TO: MVC REGLIMA,REG32MB SET REGION LIMIT ABOVE TO 32MB, via the sample below:

```
USING REGION,R07 ADDRESSABILITY FOR REGION DSECT
OI REGFLAGS,X'80' SET THAT IEFUSI CONTROLS REGIONS
TM 0(R08),X'80' V=R JOB?
BO EXIT YES USE DEFAULT VALUES
L R09,16(R0) Obtain CVT Pointer
L R11,560(R09) Obtain GDA Pointer
L R09,164(R11) Obtain Region Size below
L R11,REDUCLIM Load subtract value
SR R09,R11 Subtract by 512KB
L R10,REGSZREQ GET REQUESTED REGION SIZE
CR R09,R10 Is Requested size bigger
BNH CHNGLIM Yes do the change
LTR R10,R10 IS IT ZERO
BZ CHNGLIM Yes do the change
B EXIT Go to exit no change to req
CHNGLIM ST R9,REGLIMB Store REGLIMB to Parmlist
SR R09,R11 Subtract another 512KB
ST R09,REGSIZB Store REGSIZB to Parmlist
*
```

and add in the constants section the 512KB subtraction value:

```
REDUCLIM DC X'00080000'
```

There is also a change of the REGION SIZE BELOW value because this value is being used for programs that do a variable GETMAIN and tend to use all available storage. For an example, you can refer to the informational APAR II05315 with the title:

```
ABEND878 OR ABEND40D DURING SMP/E WITH REGION=0M OR REGION=0K OR REGION=32M WITH
NO IEFUSI
```

Disclaimer: Because only some basic function testing has been performed for the code sample in Example 2-15, extensive testing should be done to the complete EXIT functionality before it is used in a production environment.

3

Mixed language applications

In this chapter, we review how mixed language applications can access Information Management System (IMS) DB, DB2, and WebSphere MQ and how to debug mixed language applications.

The major topic covered in this chapter is language interoperability:

- How COBOL can call Java and vice versa
- How a new IMS Java batch application can interface with existing COBOL modules
- How a mixed language application can access IMS DB, DB2, and WebSphere MQ

© Copyright IBM Corp. 2014. All rights reserved.

3.1 Accessing DB2 from mixed language applications

IMS applications can access IMS DB and DB2 data. For mixed language applications each of the languages (Java and COBOL) can access DB2. IMS makes sure that the updates from all languages are within the same unit of work or transaction boundary. Furthermore, static and dynamic SQL can be mixed as required. In addition to SQL queries, it is also possible to call DB2 stored procedures.

Modernization is usually started with existing COBOL or PL/I applications that use static SQL and the traditional EXEC SQL calls that must be processed by a matching precompiler. Moving to dynamic SQL requires the users/callers authorization against the database object. This is different for static SQL, where the authorization is against the package.

For Java to use the DB2 Universal Java Database Connectivity (JDBC) Driver, the following three JAR files are required in the IMS regions class path:

- db2jcc.jar
- db2jcc_javax.jar
- db2jcc_license_cisuz.jar

and the use of a compatible connection URL, such as:

String url = “jdbc:db2os390sqlj:”;

The connection to the DB2 subsystem for the DB2 JDBC Driver is configured as Resource Recovery Services (RRS) connectivity type for Java Dependent IMS regions (JMP, JBP), and as External Subsystem Attach Facility (ESAF) connectivity type for all other Java capable IMS regions (message processing program (MPP), batch message processing (BMP)).

> **Note:** Due to restrictions in the usage, the DB2 stub linkage needs to be different for Java and non Java IMS regions.
>
> Currently, modules that have the DB2 stub statically linked can either be used in ESAF or Resource Recovery Services attachment facility (RRSAF) environments, but not in both. Modules serving the same purpose or implementing the same functionality must have different names, one for the ESAF and one for the RRSAF environment.

The connection definition for an IMS application is done by using subsystem member (SSM) definitions, which in the sample case in Example 3-1 contains two entries, one for ESAF and one for RRS.

Example 3-1 Sample IMS subsystem definition for DB2 with both ESAF and RRS

```
SST=DB2,SSN=DSNA,LIT=SYS1,ESMT=DSNMIN10,REO=R,CRC=-
SST=DB2,SSN=DSNA,COORD=RRS
```

If there is a requirement to allow the Java programmers to use dynamic SQL with JDBC but to use the authorization schema of static SQL, pureQuery using DB2 can be considered as a solution. Dynamic JDBC calls can be recorded and turned into a static package and the pureQuery runtime executes those calls as though the source code had static SQL.

For more information, refer to 2.5, “Access of IMS Java batch applications with pureQuery” on page 11.

3.2 Accessing WebSphere MQ from mixed language applications

When accessing WebSphere MQ in traditional languages both with the ESAF for MPP and BMP and with RRS for JMP and JBP, it is a matter of the definition in the IMS SSM member. The use of the WebSphere MQ Java classes currently is not supported. The WebSphere MQ Java classes lack the support for ESAF because it is not implemented. But with the use of RRS in Java Dependent IMS regions (JMP, JBP), the WebSphere MQ Java classes can be configured to work and be part of the IMS unit of work.

See Example 3-2 for a sample code that worked in a JMP. It is required to use the option MQPMO_SYNCPOINT for the WebSphere MQ calls to belong to the IMS unit of work.

Example 3-2 Making the WebSphere MQ Java classes work in IMS Java dependent regions with RRS

```
      private static final String qManager = "QM01";
      private static final String qName = "TEST.QL1";
      private static MQQueueManager qMgr = null;

           int openOptions = MQConstants.MQOO_INPUT_AS_Q_DEF |
MQConstants.MQOO_OUTPUT;

           MQQueue queue = qMgr.accessQueue(qName, openOptions);

           MQMessage msg = new MQMessage();
           msg.writeUTF("Hello, World!");

           MQPutMessageOptions pmo = new MQPutMessageOptions();
           pmo.options = MQConstants.MQPMO_SYNCPOINT;

           queue.put(msg, pmo);

           MQMessage rcvMessage = new MQMessage();

           MQGetMessageOptions gmo = new MQGetMessageOptions();
           gmo.options = MQConstants.MQGMO_ACCEPT_TRUNCATED_MSG
            + MQConstants.MQGMO_SYNCPOINT;
           gmo.matchOptions = MQConstants.MQMO_NONE;

           queue.get(rcvMessage, gmo);

           String msgText = rcvMessage.readLine();
           System.out.println("The message is: " + msgText);

           queue.close();

           qMgr.disconnect();
         }
         catch (MQException ex) {
           System.out.println("A WebSphere MQ Error occured : Completion Code " +
ex.completionCode
               + " Reason Code " + ex.reasonCode);
           ex.printStackTrace();
           for (Throwable t = ex.getCause(); t != null; t = t.getCause()) {
             System.out.println("... Caused by ");
```

```
            t.printStackTrace();
          }

        }
        catch (java.io.IOException ex) {
          System.out.println("An IOException occured whilst writing to the
message buffer: " + ex);
          ex.printStackTrace();
        }
        return;
```

See Example 3-3 for the subsystem member definition that is required to access WebSphere MQ when using RRS.

Example 3-3 Subsystem member definition to use the WebSphere MQ Java classes with RRS

```
SST=DB2,SSN=WMQA,COORD=RRS
```

In addition to the classes required for IMS Java based applications and for access to DB2, the following JAR files need to be in the IMS region's class path to make Example 3-3 work:

- com.ibm.mq.headers.jar
- com.ibm.mq.pcf.jar
- jta.jar
- connector.jar
- com.ibm.mq.commonservices.jar
- com.ibm.mq.jar
- com.ibm.mq.jmqi.jar
- com.ibm.ffdc.jar

3.3 Debugging mixed language applications

Currently, when an application starts with COBOL, the debugging for the COBOL code can be done as usual. When using the debug tool on z/OS or the GUI Eclipse plug-in, you can use the supported options to inform the debug tool about the port where the debug session is waiting.

When Java is called from COBOL to debug the Java code, there is an option that lets the JVM wait for a connected debug GUI, as is done in Eclipse or Rational Tools. When the connection is established, step-by-step execution or working with breakpoints is supported.

The JVM option to let JVM wait for a debug session from Eclipse before continuing is shown in Example 3-4.

Example 3-4 JVM option for JVM wait

```
-Xdebug
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=7777
```

This option tells JVM to turn on debug mode. That JVM is a server waiting for incoming socket connections waiting on port 7777 for a debugging session and to suspend any work until the debugging session with the Eclipse client (could also be Rational Application Developer or Rational Developer for System z for Java debugging) is established. This means that without clicking, continue in the Eclipse debug session; the JVM will not do any work.

Currently, there is no integrated view for both COBOL and Java. However, when the debug tool GUI is installed in the same Eclipse installation and the connections have been established, the Java debug view comes to the front as soon as there is a switch from COBOL to Java.

> **Restriction:** When COBOL is called by Java, the debug tool is not notified. So if there are cascading chains like **COBOL** → **Java** → **COBOL**, it is not possible to debug COBOL that is called by Java. This includes IMS Java Dependent Regions where Java is the first language that gets control.

3.3.1 Tools to debug mixed language applications

If it is required to debug a mixed language application, two different tools are required, one to debug Java and another to debug COBOL. In this section, we review the scenario where COBOL calls Java.

To debug COBOL, we used the Debug Tool for z/OS V12 (Debug Tool) and the Eclipse-based GUI component for the IBM Debug Tool Plug-in for Eclipse. A stand-alone configuration (no charge) along with IBM IMS Enterprise Suite Version 2.1.1.3 Explorer for Development (IMS Explorer) can be used for GUI-based debugging for clients that have Rational Developer for System z licensed.

To configure the Debug Tool, we did the following steps. Clients that prefer to use Rational Developer for System z can skip the first two steps:

1. Download the IBM Debug Tool Plug-in for Eclipse. Instructions can be found at:

   http://www-01.ibm.com/support/docview.wss?uid=swg24026610

2. Install the IBM Debug Tool Plug-in for Eclipse. Instructions for V12 can be found at:

   ftp://public.dhe.ibm.com/software/htp/pdtools/plugins/DT_plugin_V12100_readme.pdf

3. IMS Explorer starts and following the instructions in the link in step 2, the IBM Debug Tool Plug-in for Eclipse was installed.
4. Either restart IMS Explorer or use Rational Developer for System z directly. The latest Rational Developer for System z version usually also supports the latest version of the Debug Tool.
5. Check the remote debug port in the Preferences:
   - Open with **Window** → **Preferences**
   - Select **Run/Debug** → **Debug Daemon**
6. Switch to the debug perspective.
7. To start the Debug UI daemon on the local machine, click the small icon that looks like a red network cable, in the upper left corner (see Figure 3-1). The icon should turn green when started.

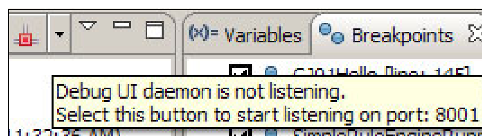


Figure 3-1 Debug UI daemon icon: Click to start the daemon

8. Prepare the COBOL application for debugging. This is documented in the Debug Tool manuals. The COBOL program needs to be compiled with specific options (for example TEST(NOSEP)) to have the debug information as part of the executable file.
9. Create the options file for debugging. See Example 3-5.

Example 3-5 Generated Debug Tool Options file

```
<PGL>CJTST*
<TST>TEST(ALL,*,PROMPT,TCPIP&172.17.36.87%8001:*)
```

For this sample, the Debug Tool panels (**Option 6 → Debug Tool User Exit Dataset**) was used. It specifies the programs and modules with wildcards for which debugging should be enabled as well as the TCPIP address and port of the Rational Developer for System z debug daemon.

> **Note:** A direct connection is required. Port forwarding is required for networks that are separated with Network Address Translation [NAT]. If the TCP/IP address of the client is not routable from z/OS, the remote debugging will not work with the described procedure. In our test environment, VPN sessions with TCP/IP addresses accessible by z/OS were successfully used.

10. A custom version of the EQAD3CXT module needs to be created. It should contain the assembler code for the initialization and termination of Language Environment enclaves and the preparation that this Language Environment enclave is enabled for debugging. It creates custom versions of CEEPIPI and CEEBINIT. This is required to support CEEPIPI-enabled execution environments for debugging with debug tool such as a JVM enabled IMS region.

    In addition to the code, the EAQAD3CXT also contains settings, such as:

    – Naming conventions for the Debug Tool Options file (for example, `&USERID.DBGTOOL.EQAUOPTS`)
    – Option for writing detailed trace output (MSGS_SW DC X'02' message level)
    – The required registration for a callback routine in a CEEPIPI environment (RRTN_SW DC X'01' register or no register rtn)

    For the assembly and the required rebinds of CEEPIPI and CEEBINIT, refer to the Debug Tool manuals at the following location:

    http://www-01.ibm.com/software/awdtools/debugtool/library

11. To be able to use the Debug Tool for debugging IMS programs, the library with the modules from step 10 needs to be placed in the IMS region steplib concatenation and the Debug Tool load library. Refer to the Debug Tool manuals for more information about how to tailor the environment.
12. Finally, start the IMS region and start the IMS transaction. The debugging session should now start, as shown in Figure 3-2 on page 47.

Because everything is generic in the preceding definitions, the debug session starts within the JVM execution. However, this is not the debug session of the JVM. The JVM debug session needs to be started manually before the COBOL application is resumed.

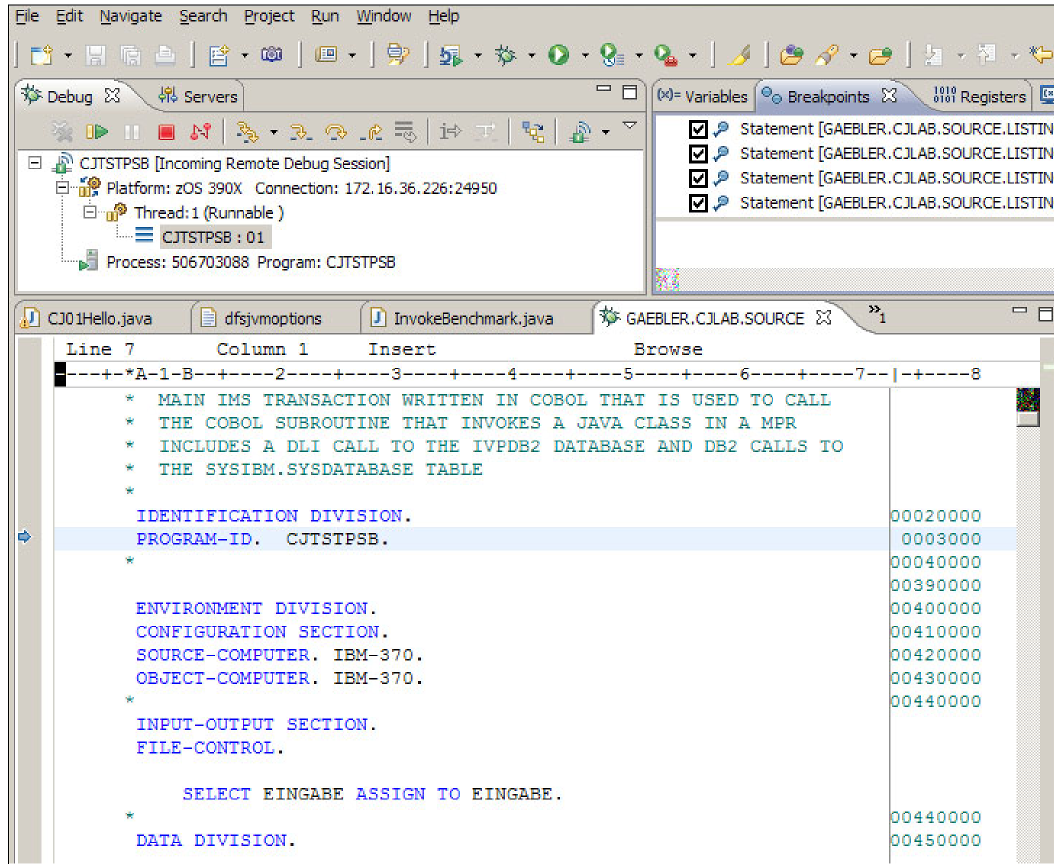


Figure 3-2 Start and suspend debug session from the IMS region

To do mixed language debugging for a COBOL application that calls Java, assuming Debug Tool, the JVM and Rational Developer for System z are configured for debugging. The following steps are required:

1. Start the IMS region.
2. Start the COBOL application.
3. The JVM stops during initialization and waits for the debug session. The job log prints this message:

```
DFSJVM00: JVM initialization started:  Tue Jul 10 08:51:02.879 2012
Listening for transport dt_socket at address: 7778
```

4. Click the highest level that is first called Java class that is to be debugged when called from COBOL (the one that contains the breakpoints), then do the following steps:
   - Select **Debug As** → **Debug Configurations**
   - Choose Remote Java Application, which has the configuration for the remote JVM (`zoshostname port 7778`)
   - Click **Debug** to connect to the JVM
5. Switch to the Debug perspective. After the JVM has been initialized, the COBOL debug session will start and suspend.

The Rational Developer for System z debug view, with both debug sessions (JVM and the Debug Tool) active but waiting at a breakpoint in COBOL, will look as shown in Figure 3-3 on page 48.

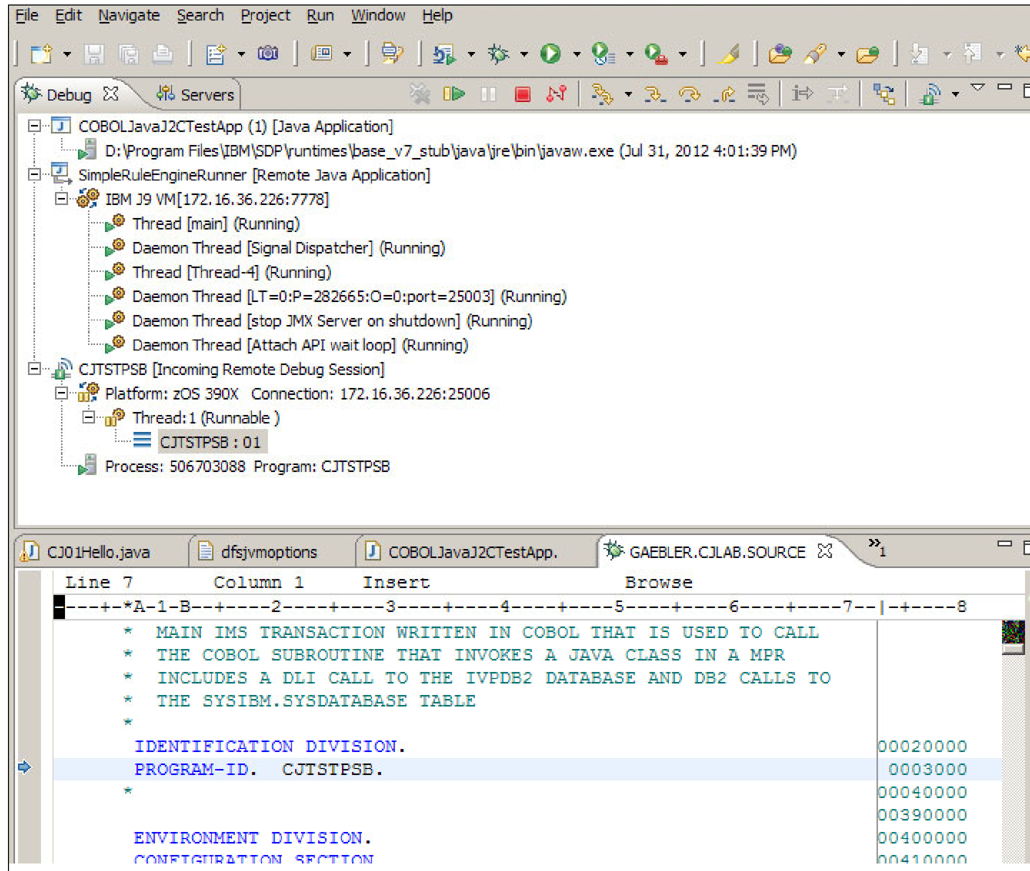


Figure 3-3 Rational Developer for System z debug perspective with both the Java and COBOL debug session active

In summary, using the latest version of the Debug Tool allows traditional language debugging in a JVM enabled IMS region and in CEEPIPI-enabled environments. The Rational Developer for System z built-in GUI can be used for remote debugging and the no-cost version based on IBM CICS Explorer® or IMS Explorer.

Java debugging is done by the Eclipse-built-in Java remote debugger.

3.4 IMS preload in a mixed environment

Preload is a performance feature that allows for the loading of modules at the time of region initialization rather than when they are first used. At the time of writing of this book, in IMS Java regions there is no preload available that would be beneficial for mixed language applications.

In IMS MPPs where a JVM can be configured to start, there is a choice to create the DLLs to be called from Java as PDSE members. One or more PGMLIB data sets are supported to be PDSEs. If the name of the DLL member is included in the DFSMPLxx member, it will be preloaded on region initialization, unlike HFS files or in IMS Java dependent regions where it is already loaded on the first use. In this way, the penalty of the first caller encountering a longer schedule to first call time can be significantly reduced, especially for larger implementations.

However, this feature is not natively available. It requires the JCL of the OO COBOL module to be configured to create a PDSE member, and it requires a manual change of the generated Java source stub file. See Example 3-6.

Example 3-6 Sample generated Java source stub file

```
// PP 5655-S71 IBM Enterprise COBOL for z/OS  4.2.0
//
// Generated Java class definition for COBOL class com.ibm.cjlab.JCOOCob
//
// Date generated: 03/09/2011
// Time generated: 07:01:26
//
//   ** Do not edit or modify this file!             **
//   ** It is (and must be) regenerated whenever the **
//   ** COBOL class is compiled.                     **
//
package com.ibm.cjlab;
public class JC01Cob
  extends java.lang.Object {
  public native void callExecute(
    java.lang.Object JAVAINTEXTO,
    java.lang.Object JAVAOUTTEXTO);
  private native static void _classInit();
  private java.nio.ByteBuffer JC01Cob_instanceData;
  private native void _instanceDataInit();
  public JC01Cob(){
    JC01Cob_instanceData=java.nio.ByteBuffer.allocateDirect(263);
    _instanceDataInit();
  }
  static {
    System.loadLibrary("JC01Cob");
    com.ibm.cjlab.JC01Cob._classInit();
  }
}
```

The System.loadLibrary call must be replaced with System.load and the // prefixed name of the PDSE member DLL that has been created by the linker instead of a .so binary file DLL that is usually placed in a hierarchic path of z/OS UNIX.

The changed Java stub file source looks like Example 3-7.

Example 3-7 Changed Java stub file source for accessing a DLL in a PDSE member

```
// PP 5655-S71 IBM Enterprise COBOL for z/OS  4.2.0
//
// Generated Java class definition for COBOL class com.ibm.cjlab.JCOOCob
//
// Date generated: 03/09/2011
// Time generated: 07:01:26
//
//   ** Do not edit or modify this file!             **
//   ** It is (and must be) regenerated whenever the **
//   ** COBOL class is compiled.                     **
//
package com.ibm.cjlab;
```

```
public class JC01Cob
  extends java.lang.Object {
  public native void callExecute(
    java.lang.Object JAVAINTEXTO,
    java.lang.Object JAVAOUTTEXTO);
  private native static void _classInit();
  private java.nio.ByteBuffer JC01Cob_instanceData;
  private native void _instanceDataInit();
  public JC01Cob(){
    JC01Cob_instanceData=java.nio.ByteBuffer.allocateDirect(263);
    _instanceDataInit();
  }
  static {
    //System.loadLibrary("JC01Cob");
    System.load("//JC01COB");
    com.ibm.cjlab.JC01Cob._classInit();
  }
}
```

The COBOL compiler manuals and the Enterprise COBOL for z/OS developers do not recommend changing the generated Java stubs.

Note: The Java stubs are generated code. If a stub is manually changed, it needs to be changed every time after it has been regenerated or recompiled.

Currently, there is no option that allows the COBOL compiler to generate the loading of the DLL. By default, it only produces the System.loadLibrary version.

4

Alternate processing options

In this chapter, we describe the need for alternate processing options driven either by external applications or by the service concept of existing Information Management System (IMS) based and external applications. We review some options that provide clients with the ability to implement their business needs with solutions that are IMS-based.

The following major topics are covered in this chapter:

- Pros and cons of using IMS Callout and WebSphere Optimized Local Adapters
- Use of WebSphere Transformation Extender within IMS
- Additional IMS options

Any solution that uses options other than subroutine calling (for example, outbound communications) might be unsuitable for large batch workloads because it might take longer than expected or what the business allows.

© Copyright IBM Corp. 2014. All rights reserved.

4.1 IMS callout to external services

If there is a requirement to call external services (for example web services or Java Platform, Enterprise Edition 2 (Java EE 2) components such as Enterprise JavaBeans (EJB beans) or message-driven beans (MDBs) from existing IMS batch applications, there are the following options:

- Synchronous calls
- Asynchronous calls
- Both synchronous and asynchronous callout
- Call DB2 stored procedures from the application
- WebSphere Transformation Extender

4.1.1 Synchronous calls

For synchronous calls, the ICAL DLI call introduced with IMS V10 can be used. It allows an IMS Callout client to wait for messages and reply to IMS (see Figure 4-1). This access pattern is synchronous, so the batch application waits for the result. The next request can only be sent after receiving the reply.

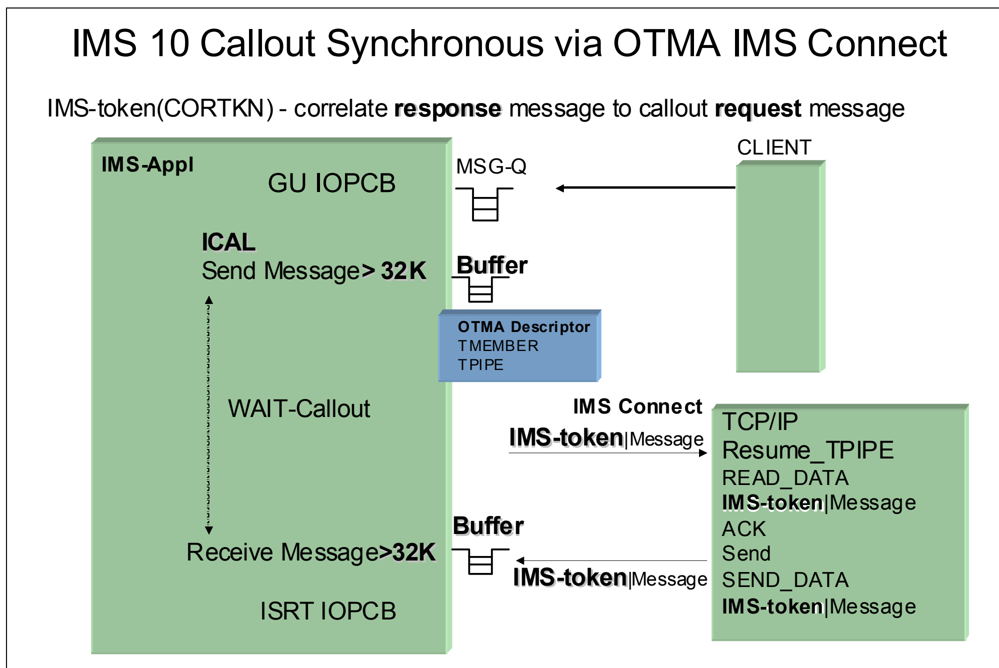


Figure 4-1 Synchronous Callout using ICAL for IMS applications

This implementation requires IMS Connect as IP Gateway for the Callout Client. Client implementations have been implemented as IMS SOAP Gateway (for SOAP requests to external Web Services), MDBs, or EJB beans deployed to a Java EE 2 application server and pure Java clients (J2SE) as plug-ins to ISV software.

For samples on IMS callout, refer to the IMS Version 11 information at the following web page:

http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp?topic=%2Fcom.ibm.ims11.doc%2Fimshome_v11.htm

4.1.2 Asynchronous calls

For asynchronous calls, queuing to the alternate PCB can be used. An IMS Callout client needs to wait for the messages; the reply to IMS is optional. See Figure 4-2.

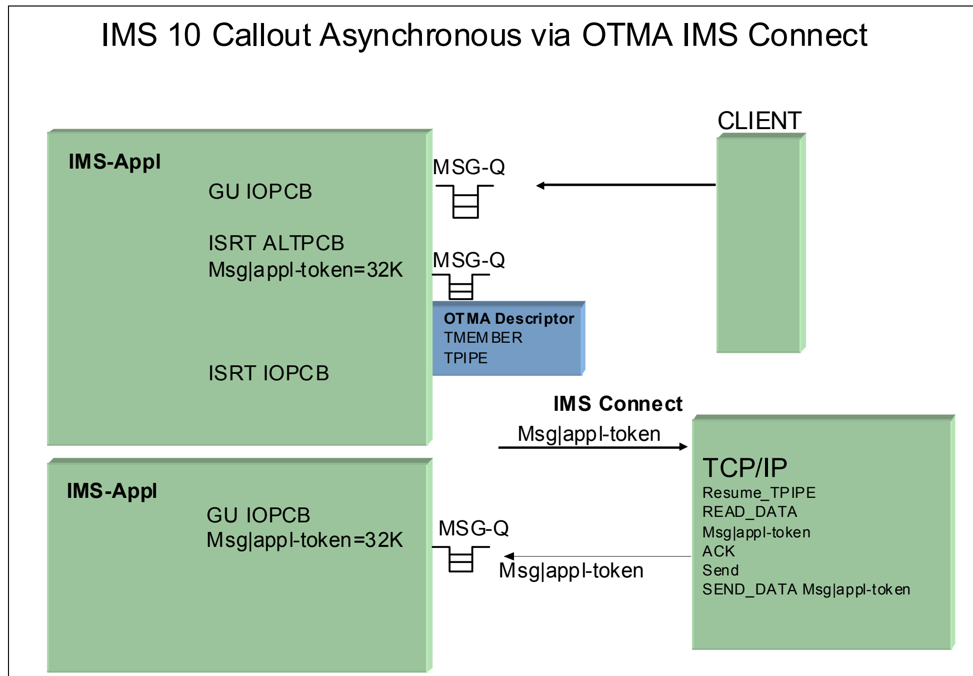


Figure 4-2 Asynchronous callout using alternate PCB for IMS applications

This pattern can also be used to queue multiple requests (for example, 1000) one after the other, and then process the replies from a callout client one after the other.

This significantly reduces the elapsed time, because the *send-wait-receive-send-wait-receive* processing pattern is transformed into a *send-send-receive-receive* processing pattern. If a link between request and response is required, the data sent back and forth should contain something like a token or correlator to correlate the request with the response.

4.1.3 Both synchronous and asynchronous callout

WebSphere MQ can implement both synchronous and asynchronous callout. The patterns are similar to ICAL and Alternate PCB-based solutions. Some clients prefer the WebSphere MQ programming interface. Figure 4-3 shows the possible use of WebSphere MQ for z/OS from IMS applications.

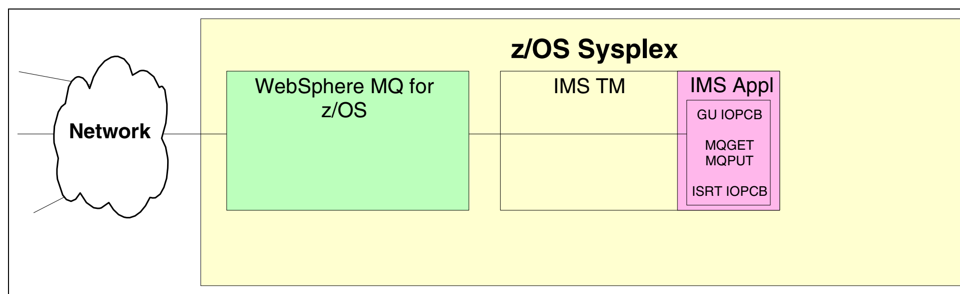


Figure 4-3 WebSphere MQ for z/OS from IMS applications

This requires the registration of WebSphere MQ as an ESAF capable subsystem in IMS and the use of the WebSphere MQ API (such as MQOPEN, MQPUT, MQGET, and MQCLOSE) within the IMS application and a local WebSphere MQ for z/OS queue manager on the z/OS image that the IMS application is running.

The patterns are implemented by using WebSphere MQ. For samples, refer to the WebSphere MQ for z/OS manuals in the IBM WebSphere MQ Information Center:

http://pic.dhe.ibm.com/infocenter/wmqv7/v7r5/index.jsp

4.1.4 Call DB2 Stored Procedures from the application

Calling DB2 Java stored procedures is an option that some clients have used to call Java methods. Basically, in DB2 for z/OS, a Java stored procedure is set up and can then easily be called from any Language Environment language caller. The calls to the DB2 Stored Procedure is always synchronous and the application waits for the result and returns from the DB2 Java Stored Procedure. The code that executes in a DB2 Java Stored procedure is part of the IMS unit of work if it does not do any outbound calls using a non-two-phase commit capable protocol.

4.1.5 WebSphere Transformation Extender

The IBM WebSphere Transformation Extender has a lot of options for calling external services. For instance, it can create a map that uses any possible target for the mapping, such as Web Service calls, or calls to external systems through J2C Resource Adapters.

For more information about WebSphere Transformation Extender calls, refer to 4.3.4, “Using data transformations in batch” on page 63. If the execution of a map that belongs to the caller’s unit of work depends on the target of the map, it is required to check if the transformation that is about to be used has transactional capabilities in all the components.

4.1.6 WebSphere z/OS Optimized Local Adapters

IBM WebSphere Optimized Local Adapters (OLA) for z/OS is an option when IMS access to services or Java programs hosted on the mainframe is required. It can also be used if the Java runtime is not implemented in IMS. See Figure 4-4 on page 55 for an overview about how WebSphere OLA for z/OS can be used from IMS applications.

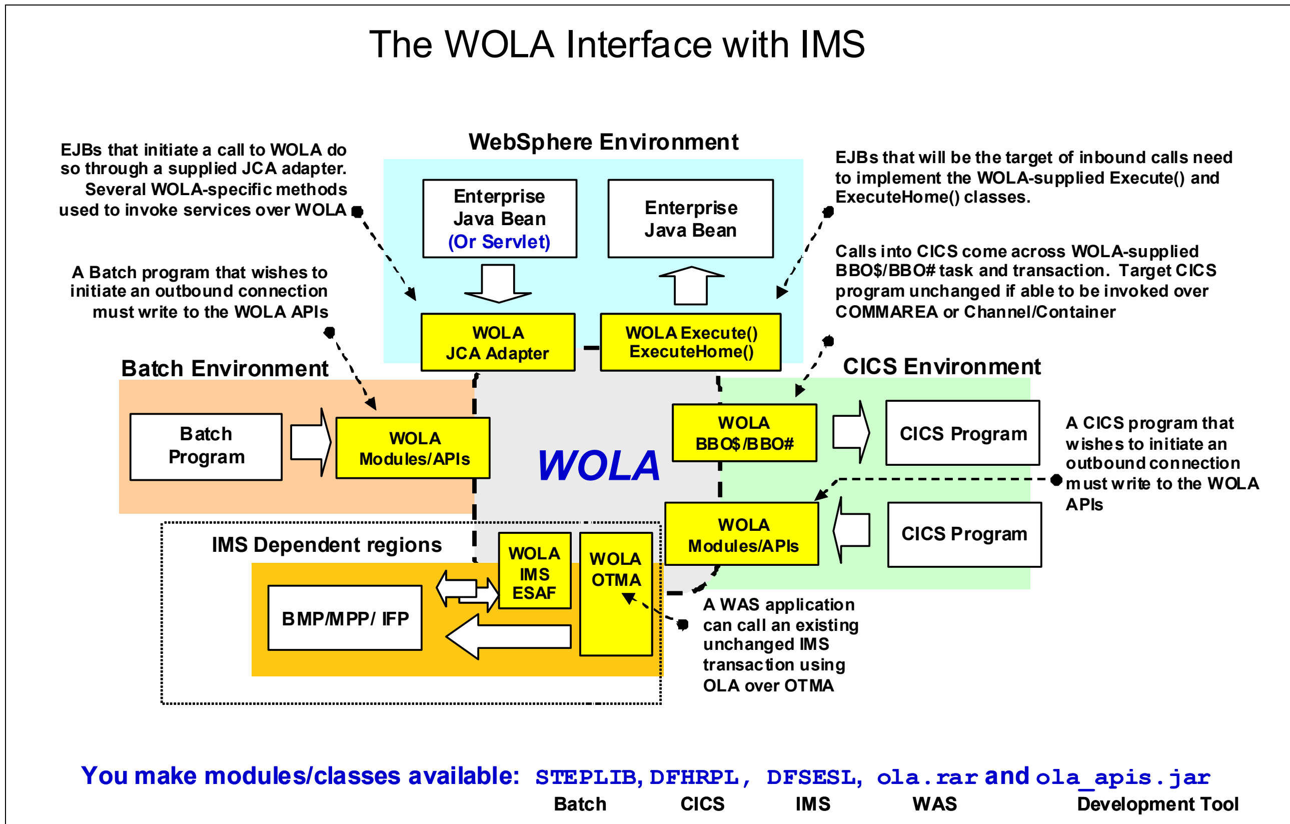


Figure 4-4 Overview of how WebSphere OLA for z/OS can be used from IMS applications

With this interface, you can implement all previously described application patterns.

Using WebSphere OLA for z/OS has the advantage of being transactional. The disadvantage is that every call to a Java method is a transaction and crosses address space boundaries, whereas COBOL/Java interoperability works more like a subroutine call.

A sample workload to call a Java method 10,000 times took 4 seconds with COBOL/Java interoperability. It took 160 seconds using WebSphere OLA for z/OS and running the Java part of the application in WebSphere z/OS on the same LPAR.

Example 4-1 is a sample subroutine that uses WebSphere OLA for z/OS to access Java and is used in IMS batch.

Example 4-1 Sample subroutine that uses WebSphere OLA for z/OS to access Java

```
IDENTIFICATION DIVISION.
       PROGRAM-ID. CALLWOLV.

       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
      *
       01   FILLER                            PIC X(32)  VALUE
            '** Working storage starts here**'.
       01 daemongroupname               PIC X(8) VALUE LOW-VALUES.
       01 daemonname                    PIC X(8) VALUE LOW-VALUES.
       01 reqtype                       PIC 9(8) COMP VALUE 0.
       01 rqst-area                     PIC X(100) VALUE SPACES.
```

```
 01 rqst-area-addr              USAGE POINTER.
 01 resp-area                   PIC X(2048) VALUE SPACES.
 01 resp-area-addr              USAGE POINTER.
 01 nodename                    PIC X(8) VALUE 'W12N01  '.
 01 servername                  PIC X(8) VALUE 'W12S01N '.
 01 registername                PIC X(12) VALUE SPACES.
 01 servicename                 PIC X(255).
 01 minconn                     PIC 9(8) COMP VALUE 1.
 01 maxconn                     PIC 9(8) COMP VALUE 10.
 01 regopts                     PIC 9(8) COMP VALUE 0.
 01 urgopts                     PIC 9(8) COMP VALUE 0.
 01 servicenamel                PIC 9(8) COMP.
 01 waittime                    PIC 9(8) USAGE BINARY.
 01 async                       PIC 9(8) USAGE BINARY.
 01 tmp-len                     PIC 9(8) COMP VALUE 0.
 01 con-handle-addr             PIC X(12) VALUE LOW-VALUES.
 01 rqst-len                    PIC 9(8) COMP VALUE 100.
 01 resp-len                    PIC 9(8) COMP VALUE 100.
 01 rc                          PIC 9(8) COMP VALUE 0.
 01 rsn                         PIC 9(8) COMP VALUE 0.
 01 rv                          PIC 9(8) COMP VALUE 0.
 01 rc-urg                      PIC 9(8) COMP VALUE 0.
 01 rsn-urg                     PIC 9(8) COMP VALUE 0.
*
 LINKAGE SECTION.
*
  01 DURCHLAEUFE                PIC S9(9) COMP VALUE +1000.
 PROCEDURE DIVISION using DURCHLAEUFE.
  Main Section.
     MOVE 'CALLWOLV     ' TO registername
     MOVE 'W12TMP ' TO daemongroupname
     MOVE 'W12TMP ' TO daemonname
     INSPECT daemonname CONVERTING ' ' TO LOW-VALUES
     MOVE 1000000  TO DURCHLAEUFE
     PERFORM INVOKEREQ THRU INVOKEREQ-END
     PERFORM BBOA1INV THRU
             BBOA1INV-END DURCHLAEUFE TIMES
     PERFORM BBOA1URG THRU BBOA1URG-END
     .
 LEAVE-CALLWOLV section.
     GOBACK
     .
  BBOA1REG Section.
     CALL 'BBOA1REG' USING daemonname, nodename, servername,
           registername, minconn, maxconn,
           regopts, rc, rsn.
     IF rc > 0 THEN
       DISPLAY "CALLWOLV: Bad RC/RSN from BBOA1REG: " rc "/" rsn
       PERFORM BBOA1URG THRU BBOA1URG-END
       GOBACK
     ELSE
       PERFORM BBOA1INV THRU BBOA1INV-END
     END-IF
     .
  BBOA1REG-END Section.
```

```
  INVOKEREQ Section.
     MOVE 0 To waittime.
     MOVE 1 TO reqtype.
     MOVE 'ejb/voidCall' TO servicename.
     INSPECT servicename CONVERTING ' ' TO LOW-VALUES.
     MOVE 0 TO servicenamel.
     MOVE 'Hello from testcase CALLWOLV!!' TO rqst-area.
     SET rqst-area-addr TO ADDRESS OF rqst-area.
*    Calculate rqst-area length
     INSPECT FUNCTION REVERSE(rqst-area)
-    TALLYING tmp-len FOR LEADING SPACES.
     COMPUTE rqst-len = LENGTH OF rqst-area - tmp-len.
     MOVE 0 TO async.
  INVOKEREQ-END Section
     .
  BBOA1INV section.
     CALL 'BBOA1INV' USING registername, reqtype,
           servicename,
           servicenamel,
           rqst-area-addr, rqst-len,
           resp-area-addr, resp-len,
           waittime,
           rc, rsn,
           rv
     IF rc > 0 THEN
       DISPLAY "CALLWOLV: Bad RC/RSN from BBOA1INV: " rc "/" rsn
       PERFORM BBOA1REG
     END-IF
     .
  BBOA1INV-END.

 BBOA1URG section.
     CALL 'BBOA1URG' USING registername, urgopts, rc-urg, rsn-urg.
 BBOA1URG-END.
 End program CALLWOLV.
```

4.2 Calling IMS transactions from traditional batch

Some clients require multiple calls to existing IMS transactions from traditional batch. The implementations discussed in this section are based on non Java solutions running on z/OS.

The following implementations have been done at client sites:

- Writing a COBOL client
- The OTMA Callable Interface (OTMA CI)
- Use DB2 Stored Procedures
- WebSphere Transformation Extender
- WebSphere MQ with its IMS OTMA Bridge

The specific order of the options is neither a recommendation nor does it represent the number of clients that used a specific solution.

4.2.1 Writing a COBOL client

Write a COBOL client (such as by modifying the old COBOL sample for IMS Connect) and perform synchronous transaction calls using that client.

Although IMS Connect is two phase commit capable, there is no way to include the IMS transaction in the batch program's unit of work.

Direct exploitation of IMS Connect SEND_ONLY and RESUME_TPIPE programming patterns also allows asynchronous (multiple sends followed by multiple receives) calling sequences. The synchronous send-wait-receive is also available.

4.2.2 The OTMA Callable Interface

The OTMA Callable Interface (OTMA CI) is a low-level interface that is similar to using WebSphere MQ or APPC calls. See Figure 4-5.

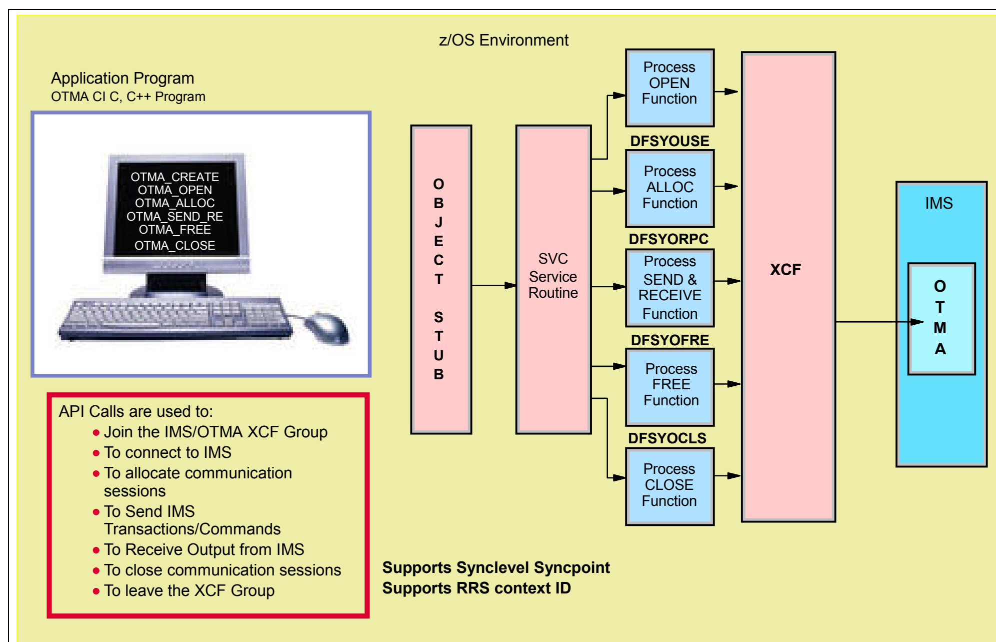


Figure 4-5 Overview of the OTMA Callable Interface Design

IMS transaction can be implemented with a simple otma_create, otma_open, otma_alloc, otma_send_receive, otma_free, and otma_close call sequence a call to one or multiple (by using multiple otma_receive calls in between). The use of an RRS token is supported, so the IMS transactions can be made part of the batch applications unit of work.

Calling multiple IMS transactions per unit of work requires as many available IMS regions as IMS transactions that should be called. The IMS regions remain occupied by the called IMS transactions until the batch applications unit of work is committed or backed out.

The OTMA Callable Interface also allows the previously described synchronous and asynchronous call patterns.

A COBOL sample that starts an IMS command through OTMA CI is shown in Example 4-2 on page 59.

Example 4-2 Excerpt from a COBOL sample

```
MAIN-RTN.
      * OTMA_CREATE
           MOVE -1 TO RET.
           MOVE -1 TO RSN1.
           MOVE -1 TO RSN2.
           MOVE -1 TO RSN3.
           MOVE -1 TO RSN4.
           MOVE 10 TO SESSIONS.
           MOVE 'COBTEST' TO OTMA-CLT-NAME.
           MOVE 'IMSA' TO OTMA-SRV-NAME.
           MOVE 'IMS12XCF' TO OTMA-GRP-NAME.
           MOVE 'TEST' TO TPIPE-PREFIX.
           CALL 'DFSYCRET' USING
                OTMA-ANCHOR,
                OTMA-RETRSN,
                ECB2,
                OTMA-GRP-NAME,
                OTMA-CLT-NAME,
                OTMA-SRV-NAME,
                SESSIONS,
                TPIPE-PREFIX.
      * OTMA_OPEN
           CALL 'DFSYOPN1' USING
                OTMA-ANCHOR,
                OTMA-RETRSN,
                ECB2,
                OTMA-GRP-NAME,
                OTMA-CLT-NAME,
                OTMA-SRV-NAME,
                SESSIONS,
                TPIPE-PREFIX.
      * WAIT FOR ECB
           CALL 'DFSYCWAT' USING BY REFERENCE ECB2.
      * OTMA_ALLOC
           MOVE 'GAEBLER' TO RACF-UID.
           CALL 'DFSYALOC' USING
                OTMA-ANCHOR,
                OTMA-RETRSN,
                SESS-HANDLE,
                TEMP-ECB,
                TRAN-NAME,
                RACF-UID,
                RACF-GID.
      * OTMA_SEND_RECEIVE
           MOVE 0 TO ECB21.
           MOVE 0 TO CTX1.
           MOVE 0 TO CTX2.
           MOVE 0 TO CTX3.
           MOVE 0 TO CTX4.
           MOVE 'LTERM' TO LTERM.
           MOVE 'MODNAME' TO MODNAME.
           MOVE LENGTH OF SEND-BUFF TO SEND-BUFF-LEN.
           MOVE LENGTH OF REC-BUFF TO REC-BUFF-LEN.
           MOVE '/DIS A   ' TO SEND-BUFF.
```

```
         CALL 'DFSYSEND' USING
              OTMA-ANCHOR,
              OTMA-RETRSN,
              ECB2,
              SESS-HANDLE,
              LTERM,
              MODNAME,
              SEND-BUFF,
              SEND-BUFF-LEN,
              BY VALUE 0,
              BY REFERENCE REC-BUFF,
              REC-BUFF-LEN,
              REC-DATA-LEN,
              BY VALUE 0,
              BY REFERENCE CONTEXT,
              ERROR-MESSAGE-TEXT.
    * WAIT FOR ECB
         CALL 'DFSYCWAT' USING BY REFERENCE ECB2.
    * OTMA_FREE
         CALL 'DFSYFREE' USING
              OTMA-ANCHOR,
              OTMA-RETRSN,
              SESS-HANDLE.
    * OTMA_CLOSE
         CALL 'DFSYCLSE' USING
              OTMA-ANCHOR,
              OTMA-RETRSN.
         GOBACK.
     PROG-END.
         EXIT.
```

4.2.3 Use of DB2 stored procedures

There are methods to call IMS transactions, for example, by using the DB2 supplied DSNAIMS and DSNAIMS2 stored procedures. Under the covers, they use the OTMA Callable Interface but, in comparison, to write an OTMA CI application, the caller just supplies the required parameters to the stored procedure and calls it with the EXEC SQL CALL procedurename statement.

The use of DB2 stored procedures with a two phase commit capable DB2 attach allows the IMS transactions to be part of the batch programs unit of work. The same rule, requiring as many available IMS regions as calls to be made to IMS transactions per unit of work, applies here.

4.2.4 WebSphere Transformation Extender

The WebSphere Transformation Extender has interfaces for J2C compliant Resource Adapters. This includes the IMS TM Resource Adapter. Simply create a map that uses an IMS transaction as target for the mapping. For more information about WebSphere Transformation Extender calls, refer to 4.3.4, “Using data transformations in batch” on page 63.

4.2.5 WebSphere MQ with its IMS OTMA Bridge

The WebSphere MQ with its IMS OTMA Bridge can also be used to start IMS transactions. This interface does not allow the IMS transaction to be part of the batch application's unit of work, but all previously described synchronous and asynchronous call patterns can be implemented.

Figure 4-6 shows that the network part can be on the same or another z/OS image.

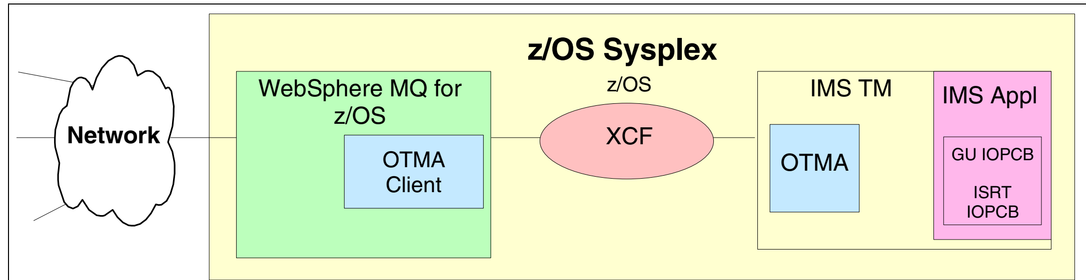


Figure 4-6 Overview of the MQ-IMS-Bridge for starting IMS transactions

For the decision on what technology, software, or architecture to use, consider operational, skills, and business needs. The options that are listed are possible implementations that some clients have used.

4.3 Accessing IMS data as result sets from traditional batch

Sometimes there is a requirement to either access IMS data from non IMS attached batch applications or to access IMS data using result sets.

However, currently there are only two options, write a DB2 stored procedure that uses the IMS Open Database Access interface (ODBA) or write some Java code. Both are then called from the batch application. The ODBA interface allows it to issue DL/I calls against IMS databases without the need of having an IMS transaction running.

There is a possibility that SQL access, which is currently available for Java methods only, could be available in a future IMS release.

ODBA applications have been implemented at client sites using COBOL, PL/I, or Java. How to write ODBA application is discussed in *IMS Connectivity in an On Demand Environment: A Practical Guide to IMS Connectivity*, SG24-6794. The ODBA caller (such as a DB2 stored procedure) needs to be at the same LPAR as the IMS system. ODBA can be used together with the Open Database Manager (ODBM) feature that was introduced with IMS V11. The use of ODBM lowers the chance of producing U113 IMS abends.

Further details about configuration are described in *IMS 11 Open Database*, SG24-7856, and in the IMS product manuals.

4.3.1 Using Business Rules Engines from batch

The IBM WebSphere IBM Operational Decision Manager Business Rules for z/OS is available for Java, and it is possible to generate rules implemented in COBOL. This solution can be a good choice for many clients.

However, some disadvantages exist:

- The rules implemented in COBOL modules must be regenerated and recompiled when a rule changes
- The rule projects for COBOL and Java cannot be shared
- Only with Java is it possible to deploy rules to the DB2 database for dynamic updating

Some clients did not want to have WebSphere for z/OS on the same LPAR as their IMS or other workloads so there is a new runtime for z/OS called *zRES*. This is a new feature for the WebSphere IBM Operational Decision Manager Business Rules for z/OS. It is implemented as a Java address space on z/OS with an API for invocation from traditional languages. A stub will be generated by the IBM Operational Decision Manager tooling that allows the traditional language to start the rules being executed in the zRES server; see Figure 4-7.

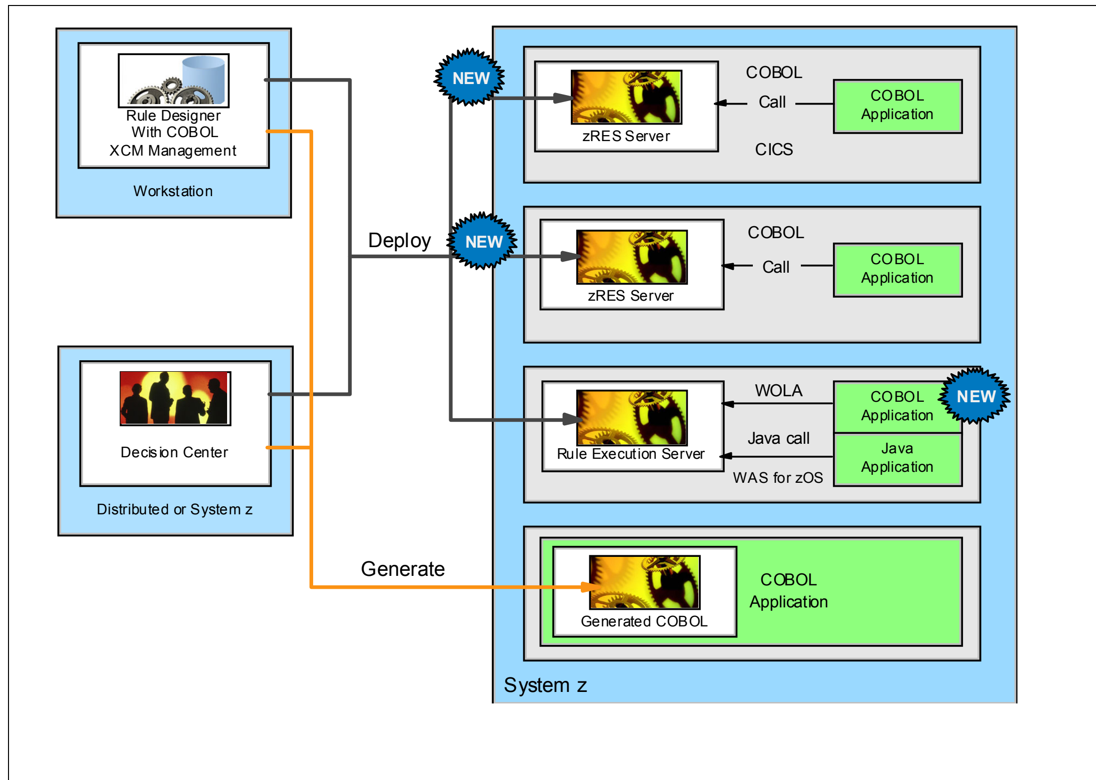


Figure 4-7 WebSphere IBM Operational Decision Manager for z/OS runtime options

In addition, there are many other Java based business rules engines on the market that can be integrated with existing batch applications by using Java interoperability and starting the rules engines as a subroutine call from the traditional languages into Java. It simply requires the rules engine to have an access pattern from a non-Java EE2 program, usually referred to as plain old Java object (POJO) or J2SE application.

4.3.2 Speeding up long running DB2 queries

Some clients have long running batch workloads that just consist of the DB2 queries. Long running in this context is a query that runs at least 10 minutes long, up to several hours.

The IBM solution to speed up those queries is DB2 Analytics Accelerator for z/OS, which is powered by IBM Netezza® technology. See the following website:

http://www-01.ibm.com/software/data/db2/zos/analytics-accelerator

This solution hooks into the DB2 optimizer and therefore is transparent to any batch workload using DB2 including IMS. There is nothing on the IMS side or in the application that must be changed that can be used to speed it up. When it is ready to work with DB2 on z/OS, IMS batch can benefit from it.

OLTP types of queries have minimal benefit from the solution because it is for long running queries. Jobs that do many updates will not benefit either. However, a possible target for the solution is batch jobs that do reporting by executing long running queries and write their output to files.

4.3.3 Speeding up calls to SQL-only DB2 stored procedures

DB2 for z/OS V9 comes with a new function called *DB2 native stored procedures*. These are SQL-only stored procedures running as threads in the DBM address space of DB2 for z/OS. That way, the application does not use the WLM scheduling, thus reducing overhead especially for many different small or short running stored procedures.

In addition, there are other benefits that might speed up a batch application. For more details, refer to *DB2 9 for z/OS Stored Procedures: The CALL and Beyond*, SG24-7604.

4.3.4 Using data transformations in batch

Today, systems interoperate with a wide variety of systems that include different data types and protocols. If there is a need to convert one format into another, there is more than one available solution.

There are IBM products on the market, such as the WebSphere Transformation Extender family, which allows you to use an Eclipse-based GUI to map one format to another. WebSphere Transformation Extender also has the ability to execute under control of an IMS region (batch and online). This allows you to execute the mapping implementations, called *MAPS*, from within IMS programs.

Refer to documentation about the IMS/DC Execution Option at the following web page:

http://pic.dhe.ibm.com/infocenter/wtxdoc/v8r3m0/topic/com.ibm.websphere.dtx.imsdc.doc/topics/g_imsdc_exec_Introduction.htm

Because WebSphere Transformation Extender is written in C and uses XPLINK(ON), it has the same restrictions on dynamic calls to subroutines as those that apply to COBOL Java interoperability.

A sample piece of COBOL code that executes a WebSphere Transformation Extender map is shown in Example 4-3 on page 64. This sample requires a map to be created and compiled with the Eclipse-based development environment that is part of the WebSphere Transformation Extender product.

Example 4-3 Sample program that calls a WTX map

```
CBL  APOST,XREF,MAP,LIST,LIB,DLL,RENT,PGMNAME(M),TEST(SYM)
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  'WTXCALL'.
      *

       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER. IBM-370.
       OBJECT-COMPUTER. IBM-370.
      *
       DATA DIVISION.
       WORKING-STORAGE SECTION.

       77  HIGH-MAP-RET-CODE     PIC 9(9)   USAGE IS BINARY VALUE ZERO.
       77  MERI-RET-CODE         PIC S9(9)  USAGE IS BINARY VALUE ZERO.

      * RUNMECBL is supplied with the IBM WS TX OS/390 SDK.
           COPY DTXRMCOB.

      *
       01  FC.
                05  FILLER              PIC X(8).
                COPY CEEIGZCT.
                05  FILLER              PIC X(4).
      *
       01  DSTX-CALL-Parameters.
           05  WC-Map-Name     Picture x(17)  value spaces.
           05  Filler          Picture x     value spaces.
           05  WC-Call-Parm1   Picture x(4)  value ' -NL'.
           05  WC-Call-Parm2   Picture x(5)  value ' -OE1'.
           05  WC-Call-Parm3   Picture x(6)  value ' -IE1S'.
           05  WC-Data-Len     Picture x(2)  value spaces.
           05  Filler          Picture x(1)  value spaces.
           05  WC-Data         Picture x(30) value spaces.
           05  Filler          Picture x     value spaces.
           05  Null-Terminator Picture xx    value low-values.

       LINKAGE SECTION.

       01  WTXRC  PIC S9(3) COMP.
       01  WTXMAP PIC X(17).
       01  WTXIN  PIC X(30).
       01  WTXOUT PIC X(30).

       77  Data-From-App       Pic x(1000).

       PROCEDURE DIVISION USING WTXRC, WTXMAP, WTXIN, WTXOUT.

      * ON ENTRY CALLER PASSES ADDRESSES FOR WTXINPUT AND OUTPUT

       MAIN-RTN.
      * Initialize the IBM Websphere Transformation Extender API
           Display 'Hallo' UPON SYSOUT.
           Call 'MercInitAPI' returning MERI-RET-CODE.
```

```
     If MERI-RET-CODE not equal 0
        then perform Init-DSTX-API-Failed.

* Now call the IBM Websphere Transformation Extender API.

* Call InitEp to initialize the ExitParm, and store the address of
* DSTX-Parameters into the ExitParm block.
     Move low-values to ExitParm.
     Move length of ExitParm to DWSIZE.
* Set address of the Map and run time options
     SET  LPDATATOAPP TO ADDRESS OF DSTX-CALL-Parameters.
     Move WTXMAP to WC-Map-Name.
     Move WTXIN to WC-Data.
     Move '30' to WC-Data-Len.

* Run the "GET" map
     Call 'RunMap' using by reference EXITPARM.
     If NRETURN  greater than   HIGH-MAP-RET-CODE
       then perform Map-Failed.

* If data was returned by the map, copy it to the old data area.
     If DWFROMLEN not equal zeroes
     then
        Set address of DATA-FROM-APP TO LPDATAFROMAPP
        Move DATA-FROM-APP to WTXOUT
     else
        Perform Map-Failed.

* expunge the null terminator if there is one.
     Inspect WTXOUT replacing all low-values by spaces.

*    Free output memory passed from DSTX
     CALL "CEEFRST" USING LPDATAFROMAPP, FC.
*
     IF CEE000 THEN
        NEXT SENTENCE
     ELSE
        DISPLAY "ERROR FREEING STORAGE FROM HEAP"
     END-IF.
*
     SET LPDATAFROMAPP TO NULL.
*
* expunge the null terminator if there is one.
     Inspect WTXOUT replacing all low-values by spaces.

* Bail Out
     GOBACK.

 Init-DSTX-API-Failed.
* IBM WS TX API initialization failed.  Write error message and
* bail out.
     Display 'IBM WS TX API initialization Failed' upon SYSOUT.
     Exit.

 Map-Failed.
```

```
      * Couldn't load map?  Write error message and bail out.
           Display 'Map ' WTXMAP  '.' upon SYSOUT.
           Display 'Return code was ' NRETURN upon SYSOUT.
           Exit.
```

To research which parameters to use, refer to the WebSphere Transformation Extender documentation at the following web page:

http://pic.dhe.ibm.com/infocenter/wtxdoc/v8r3m0/index.jsp

The sample code required to call the subroutine and pass the execution parameters from the calling program is shown in Example 4-4.

Example 4-4 Sample code used to call the WebSphere Transformation Extender map executing subroutine sample

```
            MOVE 'WTXMAPS(DENISTR)  ' TO WTXMAP.
          CALL CALLMOD USING WTXRC, WTXMAP, WTXIN, WTXOUT.
```

Refer to the WebSphere Transformation Extender manuals to get an idea about which parameters and how they are required to be specified for working in IMS. In addition, there is a possibility to preinstall the maps on IMS region start. For more information about the IMS execution option for WebSphere Transformation Extender, refer to the following document:

ftp://ftp.boulder.ibm.com/software/websphere/integration/wdatastagetx/1103.pdf

4.4 Best practices for small batches

Small batches that include Java have the impact of starting the JVM or the initialization of the batch region, which includes loading and initializing the programs. Most of the initialization of the JVM is eligible for running on a System z Application Assist Processor (zAAP) or System z Integrated Information Processor (zIIP), so there is no higher cost associated with the initialization. However, when running 1000 small batches per day, this can introduce a significant elapsed time and cost overhead plus higher CPU usage. Nevertheless, if some vital part or module is changed to use Java and is used by hundreds of batch jobs, it can make a big difference in the total runtime of sequential job networks.

4.4.1 Reduce the overhead of JVM startup

To reduce the overhead for JVM startup, it is possible to exploit the use of the JDKs Shared Classloader cache. The shared classloader cache is not only for saving I/Os when multiple JVMs load the same Java class for execution, it is also for sharing compiled versions of already executed Java classes and for the so called AOT work, which is part of the initialization of a JVM.

Batch environments based on WebSphere infrastructure use prestarted JVMs/threads, which are reused, so for this environment there are solutions to limit the overhead for small batches.

Note: At the time of writing of this book, there is no solution available for integrating WebSphere batch with IMS.

A similar infrastructure can be built with message-driven batch message processing (BMP) in IMS. Create a batch launcher infrastructure, which starts (for example) 10 IMS regions with a JVM. The BMP then waits for a message on the IMS queue about which batch program

should be started and the parameters, such as data set names for input and output files that need to be delivered. This infrastructure requires the use of dynamic allocation because it is not known in advance in what IMS region the batch workload will execute. A launcher BMP will then put a message on the IMS queue and one of the BMPs will start executing the workload. When it finishes, it puts a message on the IMS queue. The launcher can set the return code and end. That way it should be possible to integrate such batches in job networks (such as Tivoli Workload Scheduler for z/OS (TWSz)). The WebSphere infrastructure basically works the same way, but it uses xJCL to define input and output data sets, job-step sequences, return codes, selective execution, and so forth.

With the preceding approach at the expense of memory usage (prestarted BMPs or JBPs), the CPU overhead of small batches can be significantly reduced, taking advantage of the JVM shared cache functionality. However, this would be a roll-your-own (RYO) implementation not based on tooling, which at some client sites is not in policy. In most cases, standard middleware and tools should be purchased and used.

A z/OS system can have multiple JVM shared caches with different names and for different workloads. As an example, one might have two shared caches for batch workloads, two for online work (such as pure IMS Java and COBOL/Java), and one for Java workloads running outside IMS. The shared caches use common storage so that is basically one limit of the number and amount of shared JVM caches.

4.5 Summary

Most customers modernize their IMS applications as a result of business requirements and do not implement from scratch. In addition, there is a strong demand in the market to replace traditional languages in IMS applications with Java. This chapter has shown ways about how to do a migration while continuing to run the applications in IMS. The benefit is that the Java code implemented as POJO can be easily reused in other environments.

This chapter also listed some options on how to do integration (Callout) with external or non IMS based services. We also reviewed the demand of some business requirements to use transformation services such as messaging.

5

IMS batch samples

In this chapter, we provide some sample Java configurations:

- Information Management System (IMS) Java batch program
- Java configuration and IMS batch message processing (BMP) calls
- Java frameworks used with IMS Java

Software prerequisites for the solutions and samples are provided as part of this book. The solutions and samples that are described in this chapter can be easily implemented.

© Copyright IBM Corp. 2014. All rights reserved.

5.1 Sample IMS Java batch program

It is easy to run an IMS Java batch application. A sample is included with the IMS installation verification process (IVP) that can be used to get started. In this section we review the software, procedures, and configuration we used in our environment.

5.1.1 Software used in our environment

The following software was used in the test environment:

- z/OS V1R12 (initially, then migrated to z/OS V1R13)
- IMS Version 12
- DB2 for z/OS Version 10
- WebSphere MQ for z/OS Version 7.0.1
- JDK 6.0.1 for z/OS Service Release 1 (SR1)
- Enterprise COBOL for z/OS V4.2
- Enterprise PL/I for z/OS V3.6

Note: Some Java Development Kit (JDK) options are only supported with JDK 6.0.1 SR1 or later. When an older JDK is used, make sure that only supported options are used, otherwise IMS U0101 abends will occur. This means that the JVM could not be started and the error message is printed in the job log. In the case of unsupported options, it displays a message about the use of an unknown option.

5.1.2 Procedures used in our environment

To run an IMS Java batch sample program, we used the procedure listed in Example 5-1.

Example 5-1 Procedure to run IMS Java batch sample program

```
//         PROC MBR=TEMPNAME,PSB=,JVMOPMAS=,OUT=,
//              OPT=N,SPIE=0,TEST=0,DIRCA=000,
//              STIMER=,CKPTID=,PARDLI=,
//              CPUTIME=,NBA=,OBA=,IMSID=,AGN=,
//              PREINIT=,RGN=512K,SOUT=A,XPLINK=Y,
//              SYS2=,ALTID=,APARM=,ENVIRON=,LOCKMAX=
//*
//*
//JBPRGN EXEC PGM=DFSRRC00,REGION=&RGN,
//              PARM=(JBP,&MBR,&PSB,&JVMOPMAS,&OUT,
//              &OPT&SPIE&TEST&DIRCA,
//              &STIMER,&CKPTID,&PARDLI,&CPUTIME,
//              &NBA,&OBA,&IMSID,&AGN,
//              &PREINIT,&ALTID,
//              '&APARM',&ENVIRON,&LOCKMAX,&XPLINK)
//*
//STEPLIB  DD DSN=IMS12A.&SYS2.PGMLIB,DISP=SHR
//         DD DSN=IMS12A.&SYS2.SDFSJLIB,DISP=SHR
//         DD DSN=IMS12A.&SYS2.DYNALLOC,DISP=SHR
//         DD DSN=IMS12A.&SYS2.SDFSRESL,DISP=SHR
//         DD DSN=SYS1.SCEERUN,DISP=SHR
//         DD DSN=SYS1.CSSLIB,DISP=SHR
//DFSDB2AF DD DSN=IMS12A.&SYS2.SDFSRESL,DISP=SHR
//         DD DISP=SHR,DSN=DSN910.SDSNEXIT
```

```
//         DD DISP=SHR,DSN=SYS1.DSN.V910.SDSNLOD2
//         DD DISP=SHR,DSN=SYS1.DSN.V910.SDSNLOAD
//PROCLIB  DD DSN=GAEBLER.CJLAB.PROCLIB,DISP=SHR
//         DD DSN=IMS12A.&SYS2.PROCLIB,DISP=SHR
```

To run the COBOL calls for the Java sample and invoke a sample IMS Java BMP, we used the JCL listed in Example 5-2.

Example 5-2 JCL to run the COBOL calls Java sample

```
//RUNJBP   EXEC PROC=DFSJBP,
//         MBR=DFSSAM09,PSB=DFSSAM09,RGN=0M,
//         IMSID=IMSA,ENVIRON=DFSJVMEV,JVMOPMAS=DFSJVMMC
//JAVAOUT  DD  PATH='/u/gaebler/JVM.OUT',
//             PATHDISP=(KEEP,KEEP),
//             PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//             PATHMODE=SIRWXU
//JAVAERR  DD  PATH='/u/gaebler/JVM.ERR',
//             PATHDISP=(KEEP,KEEP),
//             PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//             PATHMODE=SIRWXU
//SYSPRINT DD  SYSOUT=A
//DFSSTAT   DD SYSOUT=*
```

The **ENVIRON** parameter for the environment variable settings points to the DFSJVMEV proclib member and the **JVMOPMAS** parameter for the JVM settings points to the DFSJVMMC proclib member.

5.1.3 Configuration used in our environment

The configuration and sample environment member for JVM (DFSJVMEV) that we used in our environment is shown in Example 5-3.

Example 5-3 Sample environment member for JVM (DFSJVMEV)

```
PATH=/bin:/usr/lpp/java/J601SR1/bin:.
LIBPATH=/u/gaebler:/lib:/usr/lib:/usr/lpp/java/J601SR1/bin:>
/usr/lpp/java/J601SR1/bin/j9vm:/local/db2/db2910_jdbc/lib:>
/local/ims/ims12/imsjava/classic/lib:/u/gaebler
```

There are two ways to specify the JVM options:

- Directly in the configuration member for the JVM options
- Pointing to a JVM file using the **-Xoptionsfile=** parameter as shown in Example 5-4.

Example 5-4 Sample JVM configuration member (DFSJVMMC)

```
# How to point to a XOptionsFile
#-Xoptionsfile=/u/gaebler/dfsjvmoptions
#
-Djava.class.path=     >
/u/gaebler/CJ01imsdb2.jar:/local/db2/db2910_jdbc:>
/local/db2/db2910_jdbc/classes/db2jcc.jar:>
/local/db2/db2910_jdbc/classes/db2jcc_javax.jar:>
/local/db2/db2910_jdbc/classes/db2jcc_license_cisuz.jar:>
```

```
/local/ims/ims12/imsjava/classic/imsjavaBase.jar:>
/local/ims/ims12/imsjava/classic/imsjavaTM.jar:>
/local/ims/ims12/imsjava/classic/imsJDBC.jar
#
-Xmaxf0.8
-Xminf0.3
-Xmx96M
-Xmso512k
-Xss256k
-Xms64M
#Xdebug
#Xnoagent
#Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=7777
#** Other JVM Settings
-Xcodecache10M
-Xshareclasses:name=cobolims3
#-Xshareclasses
#-Xshareclasses:printAllStats
#-Xshareclasses:verboseIO
-Xscmx64M
-Xscminaot16M
#verbose:gc
#-Xjit:verbose={compile*},verbose={options},vlog=vlog.txt
#-Xint
#-verbose:jni
-Xhealthcenter:port=1982
-Xdump:heap:events=user
```

The member contains a list of options that were used during testing. For more information and a description of possible options, refer to the Java Standard Editions website at:

http://www-03.ibm.com/systems/z/os/zos/tools/java

Note: It is possible to select the JDK level, however, IMS applications only work with 31-bit. For more information and a description of possible options, refer to the *SDK Guide* at:

http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/topic/com.ibm.java.doc.user.aix32.60/pdf/sdkguide.aix32.pdf

For mapping between the PSB and the Java class name, the DFSJVMAP member is used. The package and the class name with the main() method are specified for a given PSB as shown in Example 5-5.

Example 5-5 Sample DFSJVMAP member

```
JC01TRAN=com/ibm/cjlab/JC01JavaCobol
JPAPSB=com/ibm/ims/hibernate/sample/TestExample
DFSSAM09=sample/imsjpa/TestBMP
```

With the JCL and PROCLIB members we have listed in this section, it is possible to execute all of the Java samples by changing the DFSJVMAP member to the correct package and class name. It is also possible to test multiple Java applications with the same PSB because the PSB contains all the required PCBs. There is also a plan with the PSB name in DB2, which has the authority to do everything in DB2 that the program is written to do.

5.2 Sample Java configuration and IMS BMP calls

In this section we provide the following samples:

- Java configuration for an IMS BMP
- IMS BMP COBOL calls Java
- IMS BMP PL/I calls a Java method

5.2.1 Sample Java configuration for an IMS Batch Message Program

IMS BMPs are the way to execute a program that starts with a traditional language such as COBOL, PL/I, or Assembler. Starting with IMS V10, they can be configured to start a JVM and allow traditional programs to invoke Java methods. A sample configuration required to start the JVM in a traditional IMS BMP is shown in Example 5-6.

Example 5-6 Sample environment member for the JVM (DFSJVMEV)

```
PATH=/bin:/usr/lpp/java/J601SR1/bin:.
LIBPATH=/u/gaebler:/lib:/usr/lib:/usr/lpp/java/J601SR1/bin:>
/usr/lpp/java/J601SR1/bin/j9vm:/local/db2/db2910_jdbc/lib:>
/local/ims/ims12/imsjava/classic/lib:/u/gaebler
```

There are two ways to specify the JVM options:

- Directly in the configuration member for the JVM options
- Pointing to a JVM file using the **-Xoptionsfile=** parameter

A sample JVM configuration member (DFSJVMMC) member is shown in Example 5-7.

Example 5-7 Sample JVM configuration member

```
# How to point to a XOptionsFile
#-Xoptionsfile=/u/gaebler/dfsjvmoptions
#
-Djava.class.path=     >
/u/gaebler/CJ01imsdb2.jar:/local/db2/db2910_jdbc:>
/local/db2/db2910_jdbc/classes/db2jcc.jar:>
/local/db2/db2910_jdbc/classes/db2jcc_javax.jar:>
/local/db2/db2910_jdbc/classes/db2jcc_license_cisuz.jar:>
/local/ims/ims12/imsjava/classic/imsjavaBase.jar:>
/local/ims/ims12/imsjava/classic/imsjavaTM.jar:>
/local/ims/ims12/imsjava/classic/imsJDBC.jar
#
-Xmaxf0.8
-Xminf0.3
-Xmx96M
-Xmso512k
-Xss256k
-Xms64M
#Xdebug
#Xnoagent
#Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=7777
#** Other JVM Settings
-Xcodecache10M
-Xshareclasses:name=cobolims3
#-Xshareclasses
```

```
#-Xshareclasses:printAllStats
#-Xshareclasses:verboseIO
-Xscmx64M
-Xscminaot16M
#verbose:gc
#-Xjit:verbose={compile*},verbose={options},vlog=vlog.txt
#-Xint
#-verbose:jni
-Xhealthcenter:port=1982
-Xdump:heap:events=user
```

Restriction: In the sample JVM settings member, a continued line cannot exceed 255 characters. If that happens, the **-Xoptionsfile** parameter together with a file in the UNIX file system of z/OS needs to be used.

The member contains a list of options that were used during testing. For more information about these options and a description of all the possible options, refer to Java Standard Editions website at the following link:

http://www-03.ibm.com/systems/z/os/zos/tools/java

It is possible to pick and choose the JDK level that is used; 31-bit only works with IMS applications. The SDK Guide reviews all the possible options.

The only other change to existing IMS BMP JCL is to add the **ENVIRON=** parameter with the member containing the environment settings and the **JVMOPMAS=** parameter pointing to the name of the member containing the JVM settings.

5.2.2 Sample application IMS BMP COBOL calls Java

A sample BMP that calls two Java methods is provided as an IMS sample in this book.

The structure of the sample program where COBOL calls Java is illustrated in Figure 5-1 on page 75.

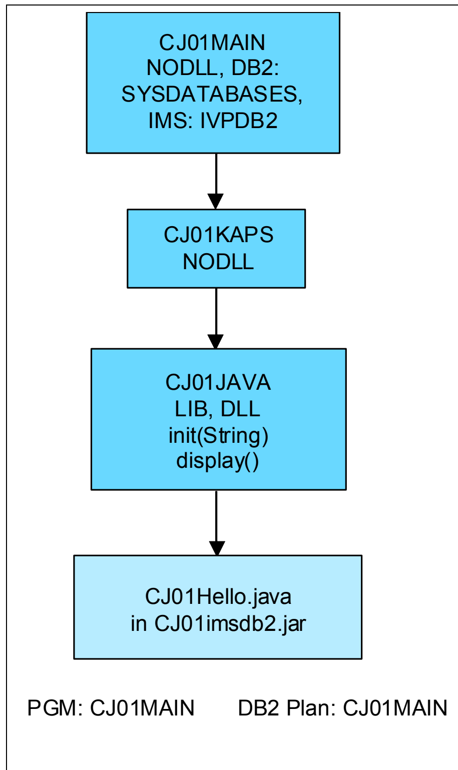


Figure 5-1 Illustration of sample program where COBOL calls Java

This sample program shows that:

- CJ01MAIN is traditionally NODLL compiled.
- CJ01JAVA needs to be DLL compiled to call Java.
- To be able to call CJ01JAVA dynamically from CJ01MAIN, it needs to have a wrapper.
- DLL modules cannot dynamically call NODLL modules and vice versa.
- But DLL and NODLL modules can be statically linked together.

Solution that is depicted in the sample program:

- CJ01KAPS is NODLL compiled and statically linked together with DLL compiled module CJ01JAVA.
- CJ01MAIN is NODLL compiled and can then dynamically call CJ01KAPS.

The JCL in Example 5-8 was used to run the sample.

Example 5-8 JCL to run the COBOL calls Java sample

```
//RUNJBP   EXEC PROC=IMSBATCH,
//         MBR=CJ01MAIN,PSB=CJ01BMP,RGN=0M,SSM=SSM,PRLD=DC,
//         IMSID=IMSA,ENVIRON=DFSJVMEV,JVMOPMAS=DFSJVMMC
//JAVAOUT  DD  PATH='/u/gaebler/JVM.OUT',
//             PATHDISP=(KEEP,KEEP),
//             PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//             PATHMODE=SIRWXU
//JAVAERR  DD  PATH='/u/gaebler/JVM.ERR',
//             PATHDISP=(KEEP,KEEP),
//             PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//             PATHMODE=SIRWXU
//SYSPRINT DD  SYSOUT=A
```

```
//DFSSTAT   DD SYSOUT=*
```

The **ENVIRON** parm pointed to member DFSJVMEV, which is listed in Example 5-9.

Example 5-9 Environment member used to run the COBOL calls Java sample

```
PATH=/bin:/usr/lpp/java/J6.0.1/bin:.
LIBPATH=/lib:/usr/lib:/usr/lpp/java/J6.0.1/bin:>
/usr/lpp/java/J6.0.1/bin/j9vm:/local/db2/db2910_jdbc/lib:>
/local/ims/ims12/imsjava/lib:/u/gaebler
```

The JVM configuration member parameter **JVMOPMAS** points to member DFSJVMMC. The JVM configuration member that is used to run the COBOL calls Java sample is listed in Example 5-10.

Example 5-10 JVM configuration member used to run the COBOL calls Java sample

```
-Xoptionsfile=/u/gaebler/dfsjvmoptions
```

It points to a file in the hierarchical file system managed by z/OS, which allows class paths longer than 255 bytes. The actual JVM options file to run the COBOL calls Java sample is listed in Example 5-11.

Example 5-11 JVM configuration file used to run the COBOL calls Java sample

```
-Djava.class.path=/u/gaebler/CJ01imsdb2.jar:/local/db2/db2910_jdbc:/local/db2/db29
10_jdbc/classes/db2jcc.jar:/local/db2/db2910_jdbc/classes/db2jcc_javax.jar:/local/
db2/db2910_jdbc/classes/db2jcc_license_cisuz.jar:/local/ims/ims12/imsjava/imsudb.j
ar:/local/ims/ims12/imsjava/imsutm.jar
#
-Xmaxf0.8
-Xminf0.3
-Xmx64M
-Xmso512k
-Xss256k
-Xms32M
-Xcodecache10M
-Xshareclasses:name=cobolims1
-Xscmx64M
-Xscminaot16M
```

The NODLL main IMS application is compiled by using the JCL in Example 5-12.

Example 5-12 Sample JCL to compile and link the main IMS transaction

```
//CJO1MAIN EXEC DSNHICOB,MEM=CJ01MAIN,USER=&SYSUID,
//             PARM.COB=RENT,REGION=1400K,
//             PARM.LKED='RENT,LIST,XREF,LET,MAP'
//PC.SYSIN    DD DSN=&SRCHLQ..SOURCE(CJ01MAIN),DISP=SHR
//PC.SYSLIB   DD DUMMY
//LKED.SYSLMOD DD DSN=&IMSHLQ..PGMLIB(CJ01TRAN),DISP=SHR
//LKED.SYSLIB  DD DSN=&IMSHLQ..SDFSRESL,DISP=SHR
//             DD DSN=&DSNHLQ..SDSNLOAD,DISP=SHR
//             DD DSN=&CEEHLQ..SCEELKED,DISP=SHR
//             DD DSN=&CEEHLQ..SCEELKEX,DISP=SHR
//             DD DSN=&SYS1HLQ..CSSLIB,DISP=SHR
//LKED.SYSIN DD *
```

```
     INCLUDE SYSLIB(DSNTIAR)
     INCLUDE SYSLIB(DFSLI000)
     ENTRY CJ01MAIN
     NAME CJ01TRAN(R)
//*
```

It produces the main routine. The COBOL application that does the JNI calls and actually invokes the Java method is compiled with the JCL shown in Example 5-13.

Example 5-13 Sample JCL to compile and link the Java method invoking routine

```
//CJ01JAVA EXEC IGYWCPL,PARM.COBOL='RENT',REGION=1400K,
//              PARM.LKED='LIST,XREF,LET,MAP,DYNAM(DLL),CASE(MIXED)'
//COBOL.SYSIN    DD DSN=&SRCHLQ..SOURCE(CJ01JAVA),
//              DISP=SHR
//* Dataset containing JNI.cpy
//COBOL.SYSLIB DD DISP=SHR,DSN=&SRCHLQ..SOURCE
//LKED.SYSLMOD DD  DSNAME=&&MODULES(CJ01JAVA),UNIT=SYSALLDA,
//              DISP=(MOD,PASS),SPACE=(TRK,(3,3,5)),
//              DCB=(BLKSIZE=3200)
//LKED.SYSLIB  DD DSN=&IMSHLQ..SDFSRESL,DISP=SHR
//              DD DSN=&CEEHLQ..SCEELKED,DISP=SHR
//PLKED.SYSIN  DD
//              DD PATH='/usr/lpp/java/J6.0/bin/j9vm/libjvm.x'
//              DD PATH='/usr/lpp/cobol/lib/igzcjava.x'
//LKED.SYSIN DD *
     INCLUDE '/usr/lpp/java/J6.0/bin/j9vm/libjvm.x'
     INCLUDE '/usr/lpp/cobol/lib/igzcjava.x'
     ENTRY CJ01JAVA
     NAME CJ01JAVA(R)
/*
```

The wrapper to call the DLL subroutine, which is required to invoke Java, the JVM is DLL compiled is shown in Example 5-14.

Example 5-14 Sample JCL to compile and link the DLL wrapper routine

```
//CJ01KAPS EXEC IGYWCL,PARM.COBOL='RENT,NODLL,NODYNAM',REGION=1400K,
//              PARM.LKED='LIST,XREF,LET,MAP'
//COBOL.SYSIN DD DSN=&SRCHLQ..SOURCE(CJ01KAPS),
//              DISP=SHR
//LKED.SYSLMOD DD DSN=&IMSHLQ..PGMLIB(CJ01KAPS),DISP=SHR
//LKED.SYSLIB  DD DSN=&IMSHLQ..SDFSRESL,DISP=SHR
//              DD DSN=&CEEHLQ..SCEELKED,DISP=SHR
//              DD DSNAME=&&MODULES,DISP=(OLD,DELETE)
//LKED.SYSIN DD *
     NAME CJ01KAPS(R)
/*
```

A PSB (CJ01MAIN in case of the sample used here) is required to run the sample and a DB2 Plan with the name of the PSB.

5.2.3 Sample application IMS BMP PL/I calls a Java method

This book provides a sample PL/I program and JCL showing how to invoke a Java method.

The main difference for PL/I in comparison to COBOL is that PL/I distinguishes between main and sub modules. Since the JVM in IMS regions is brought up as part of a CEEPIPI environment, only PL/I subs are allowed in JVM-enabled IMS dependent regions. This applies also to the “main” program.

This means that current PL/I mains will fail to execute in an IMS region that is configured to include a JVM. Alternatively, “main” PL/I programs that are compiled as subroutine modules will fail to execute in an IMS region without the JVM. The Language Environment enclave will be built by the application and therefore is required to be a PL/I main.

In our test environment, we created the same PL/I module that is the starting point to execute a PL/I program and two different versions of the module:

- As a sub stored in load library IMS.PGMLIB.PLISUB
- As a PL/I main stored in load library IMS.PGMLIB.PLIMAIN

Now, all non-JVM IMS regions point to IMS.PGMLIB.PLIMAIN and all JVM-enabled IMS regions point to IMS.PGMLIB.PLISUB. In IMS online environments, this allows workloads to move between different IMS regions and to enable a transaction class with IMS enabled regions to process transactions that do not need the JVM, which could be necessary in case of peak workloads.

The sample code included in this book, which can run in an IMS JVM enabled region and invokes a Java method, contains a `readme` file.

5.2.4 Sample application IMS Java Batch Program Java calls COBOL

Part of the IMS samples that are provided for download as part of the book is a sample Java Batch Program (JBP) that calls a COBOL routine. The structure of the sample program where Java calls COBOL is illustrated in Figure 5-2 on page 79.

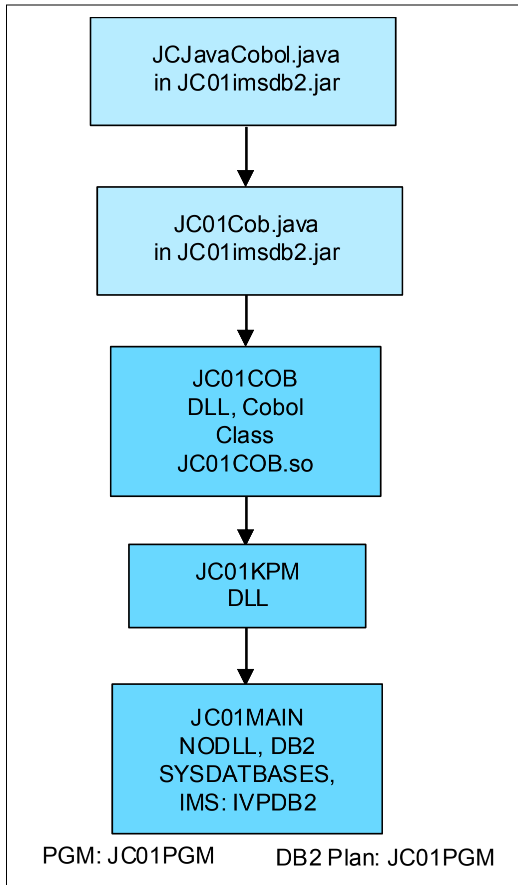


Figure 5-2 Illustration of a sample program where Java calls COBOL

This sample program shows that:

- JCJavaCobol.java is Java Main program that can run as JBP or JMP.
- JC01COB needs to be DLL compiled and an OO COBOL object to be called from Java and in order for the COBOL compiler to generate JC01Cob.java and the JC01Cob.so DLL for use from the Java JNI.
- JC01MAIN is traditionally NODLL compiled and used to run as a subroutine for an IMS transaction.
- To be able to call JC01MAIN dynamically from JC01COB, it needs to have a wrapper.
- JC01KPM is a NODLL compiled module and statically linked with JC01MAIN.
- DLL modules cannot dynamically call NODLL modules and vice versa.
- But DLL and NODLL modules can be statically linked together.

Solution depicted in sample:

- JC01COB is DLL compiled and dynamically calls JC01KPM, which is also DLL compiled.
- JC01MAIN is NODLL compiled and statically linked together with DLL compiled module JC01KPM.

The JCL in Example 5-15 was used to run the sample.

Example 5-15 JCL to run the IMS Java Batch Program

```
//JC01JMP  JOB ,'',
//         MSGCLASS=A,TIME=1440,
//         MSGLEVEL=(1),REGION=0M
//IMSPROC JCLLIB ORDER=IMS.PROCLIB
//RUNJMP   EXEC PROC=DFSJMP,
//         AGN=IVP,               AGN NAME
//         NBA=40,
//         OBA=20,
//         TLIM=10,               MPR TERMINATION LIMIT
//         SOD=,                  SPIN-OFF DUMP CLASS
//         IMSID=IMSD,
//         CL1=001,
//         ENVIRON=DFSJVMEV,      JMP ENVIRON MEMBER
//         JVMOPMAS=DFSJVMMC       JMP MASTER MEMBER
//JAVAOUT  DD  PATH='/u/gaebler/JJVM.OUT',
//             PATHDISP=(KEEP,KEEP),
//             PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//             PATHMODE=SIRWXU
//JAVAERR  DD  PATH='/u/gaebler/JJVM.ERR',
//             PATHDISP=(KEEP,KEEP),
//             PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//             PATHMODE=SIRWXU
//SYSPRINT DD  SYSOUT=A
/*
```

Similar to running Java only applications, the environment and JVM configuration settings as well as the mapping configuration to map a PSB name to a Java package and class is required. See Example 5-16 for a sample environment member for the JVM (DFSJVMEV).

Example 5-16 Sample environment member for the JVM

```
PATH=/bin:/usr/lpp/java/J601SR1/bin:.
LIBPATH=/u/gaebler:/lib:/usr/lib:/usr/lpp/java/J601SR1/bin:>
/usr/lpp/java/J601SR1/bin/j9vm:/local/db2/db2910_jdbc/lib:>
/local/ims/ims12/imsjava/classic/lib:/u/gaebler
```

There are two ways of specifying the JVM options:

- Directly in the configuration member for the JVM options
- Pointing to a JVM file using the **`-Xoptionsfile=`** parameter.

A sample JVM configuration member (DFSJVMMC) is shown in the comment in Example 5-17.

Example 5-17 Sample JVM configuration member (DFSJVMMC)

```
# How to point to a XOptionsFile
#-Xoptionsfile=/u/gaebler/dfsjvmoptions
#
-Djava.class.path=     >
/u/gaebler/CJ01imsdb2.jar:/local/db2/db2910_jdbc:>
/local/db2/db2910_jdbc/classes/db2jcc.jar:>
/local/db2/db2910_jdbc/classes/db2jcc_javax.jar:>
```

```
/local/db2/db2910_jdbc/classes/db2jcc_license_cisuz.jar:>
/local/ims/ims12/imsjava/classic/imsjavaBase.jar:>
/local/ims/ims12/imsjava/classic/imsjavaTM.jar:>
/local/ims/ims12/imsjava/classic/imsJDBC.jar
#
-Xmaxf0.8
-Xminf0.3
-Xmx96M
-Xmso512k
-Xss256k
-Xms64M
#Xdebug
#Xnoagent
#Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=7777
#** Other JVM Settings
-Xcodecache10M
-Xshareclasses:name=cobolims3
#-Xshareclasses
#-Xshareclasses:printAllStats
#-Xshareclasses:verboseIO
-Xscmx64M
-Xscminaot16M
#verbose:gc
#-Xjit:verbose={compile*},verbose={options},vlog=vlog.txt
#-Xint
#-verbose:jni
-Xhealthcenter:port=1982
-Xdump:heap:events=user
```

Restriction: In the sample JVM settings member, a continued line cannot exceed 255 characters. If that applies, the **`-XOptionsfile`** parameter together with a file in the UNIX file system of z/OS needs to be used.

The member contains a list of options that were used during testing. For more information about these options and a description of all the possible options, refer to the Java Standard Editions website at the following link:

http://www-03.ibm.com/systems/z/os/zos/tools/java

It is possible to pick and choose the JDK level that is used; 31-bit only works with IMS applications. The SDK Guide contains and describes all the possible options.

For the mapping between the PSB and the Java class name, the DFSJVMAP member is used, where the package and the class name with the main() method are being specified for a given PSB. See Example 5-18.

Example 5-18 Sample DFSJVMAP member

```
JC01TRAN=com/ibm/cjlab/JC01JavaCobol
JPAPSB=com/ibm/ims/hibernate/sample/TestExample
DFSSAM09=sample/imsjpa/TestBMP
```

To execute the native library through the JNI, a wrapper class is required. With Enterprise COBOL for z/OS, the COBOL compiler will generate the class at compile time.

The JCL used to compile the OO COBOL class into a DLL residing in the z/OS UNIX file system and generate the JNI stub Java source is shown in Example 5-19. The generated JNI stub is also shown in this example.

Example 5-19 JCL to compile and link the OO COBOL class with the generation of the JNI stub

```
//JC01COB EXEC IGYWCL,
//             PARM.COBOL='RENT,PGMN(LM),DLL,EXPORTALL',REGION=1400K,
//             PARM.LKED='RENT,LIST,XREF,LET,MAP,DYNAM(DLL),CASE(MIXED)'
//COBOL.SYSIN    DD DSN=&SRCHLQ..SOURCE(JC01COB),
//              DISP=SHR
//* Dataset containing JNI.cpy
//COBOL.SYSLIB DD DISP=SHR,DSN=&SRCHLQ..SOURCE
//LKED.SYSLIB  DD DSN=&IMSHLQ..SDFSRESL,DISP=SHR
//              DD DSN=&DSNHLQ..SDSNLOAD,DISP=SHR
//              DD DSN=&CEEHLQ..SCEELKED,DISP=SHR
//              DD DSN=&CEEHLQ..SCEELKEX,DISP=SHR
//              DD DSN=&SYS1HLQ..CSSLIB,DISP=SHR
//LKED.SYSIN DD *
     INCLUDE '/usr/lpp/java/J6.0/bin/j9vm/libjvm.x'
     INCLUDE '/usr/lpp/cobol/lib/igzcjava.x'
/*
//COBOL.SYSJAVA DD PATH='/u/gaebler/JC01Cob.java',
//             PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//             PATHMODE=(SIRWXO,SIRWXG,SIRWXU),
//             FILEDATA=TEXT
//LKED.SYSLMOD DD PATH='/u/gaebler/libJC01Cob.so',
//             PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//             PATHMODE=(SIRWXO,SIRWXG,SIRWXU)
//LKED.SYSDEFSD DD PATH='/u/gaebler/libJC01Cob.x',
//             PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//             PATHMODE=(SIRWXO,SIRWXG,SIRWXU)
```

Refer to Example 5-20 on page 86 for a description about how to use a DLL that is not in the UNIX file system of z/OS. This might be allowed with a possible future IMS function to preinstall the DLL modules into the IMS region.

In the sample application, it is also shown how to call an existing NODLL-compiled COBOL module. As for COBOL calls to Java, a wrapper module is required. The wrapper module JC01KPM is DLL compiled and statically linked with the JC01MAIN subroutine which represents the existing NODLL routine in the example scenario. With this approach, JC01KPM can be called dynamically and changed without recompile of JC01COB. This would also result in a regeneration of the JNI stub Java source and lead to a complete rebuilt of the JAR file for the Java part of the application. The JBP can then be executed.

5.3 Sample Java frameworks used with IMS Java

In this section we review how to install the required plug-ins into Rational Developer for System z. We also provide a detailed sample for using Hibernate as Object Relational (OR) mapper with IMS DB.

As part of this book we provide the most significant samples delivered in a single Rational Developer for System z workspace.

5.3.1 How to install all required plug-ins into Rational Developer for System z

It is now possible to install many of the required components for interaction with IMS resources into the same Rational Developer for System z installation. This provides the ability to do almost everything from within the same Eclipse installation.

Prerequisite for an installation of Rational Developer for System z V8.0, the following components were used:

- Enterprise Suite IMS Explorer for developers V2.1
- Enterprise Suite IMS DL/I Model Utility plug-in V2.1
- Debug Tool Plug-in for Eclipse V11.1.0.0

The Enterprise Suite components can be downloaded from the IMS home page: http://www.ibm.com/ims. There are links pointing to the Enterprise Suite and downloads.

If Rational Developer for System z is already installed, the much smaller installation files for installation on top of Rational Developer for System z can be used.

The instructions for downloading and installing the IBM Debug Tool Plug-in for Eclipse can be found at the following link:

http://www-01.ibm.com/support/docview.wss?uid=swg24026610

The IMS Enterprise Suite V2.1 components are installed using IBM Installation Manager:

- Add the compressed files as a repository and then click **install**.
- During the installation process, Installation Manager searches the repository to find the IMS Enterprise Suite Explorer for Development and the IMS Enterprise Suite DLIModel utility plug-in.
- Check these two components and continue the installation process. If older versions are already installed, they must be removed first.
- Both components were installed in the same package group as Rational Developer for System z. The default is to install them in a different package group.

Per the instructions, the Debug Tool Plug-in for Eclipse V11.1.0.0 must be installed from within the started Rational Developer for System z workspace. Go through the instructions that start with the menu option **Help** → **Software Updates** and finish the installation process.

In the end, all the perspectives offered by these components will be seamlessly integrated with the existing Rational Developer for System z installation. Currently, it is not possible to update them over the network. Updates must be applied manually.

5.3.2 Download the Hibernate JARs into the sample workspace

Currently, the workspace with the Hibernate samples does not contain all the JAR files that are required by Hibernate.

The following list of projects and JAR files are required to make the Hibernate_pure and the Hibernate_local_cache workspaces work. After download, they must be added to the class path of every Hibernate related project:

- Hibernate 3.6.8.FINAL from Hibernate.org, extract the following:
  - Executable JAR files from the hibernate-distribution-3.6.8

  - Final directory: antlr-2.7.6.jar, cglib-2.2.jar, commons-collections-3.1.jar, dom4j-1.6.1.jar, hibernate3.jar, hibernate-jpa-2.0-api-1.0.1.Final.jar, javassist3.12.0.GA.jar, jta-1.1.jar, slf4j-api-1.6.1.jar
- SLF4J 1.6.1 from slf4j.org go to previous versions
  - Select 1.6.1
  - Extract the following Executable JAR file: `slf4j-log4j12-1.6.1.jar`
- Apache Log4J 1.2 from logging.apache.org/log4j,
  - Extract the following Executable JAR file: `log4j-1.2.16.jar`

In addition to the preceding JAR files, the `Hibernate_distributed_cache` requires the downloads for EHCache.

5.3.3 Hibernate as Object Relational mapper with IMS DB

In this section, we describe an example about how to map an existing IMS Database against an object hierarchy using Hibernate. Because Hibernate only supports SQL, the use of the IMS JDBC Driver is required.

> **Note:** IMS DB does not support the CREATE TABLE or ALTER TABLE statements.

Scenario with Parts Order database

In our scenario, we use the Parts Order sample database that is provided in the IMS installation verification program (IVP).

A chart of the DI21PART IMS sample database that comes with IMS is depicted in Figure 5-3 on page 85. It can be installed and used by any IMS installation.

The chart depicts:

- The hierarchical structure of the segments in the Parts Order database.
- Each rectangle represents a database segment.
- PARTROOT is the root segment of this database, and STANINFO and STOKSTAT are its child segments.
- CYCCOUNT and BACKORDR are the child segments for STOKSTAT.
- Each segment contains one or more fields that contain data. For example, PARTKEY is a field in the PARTROOT segment.

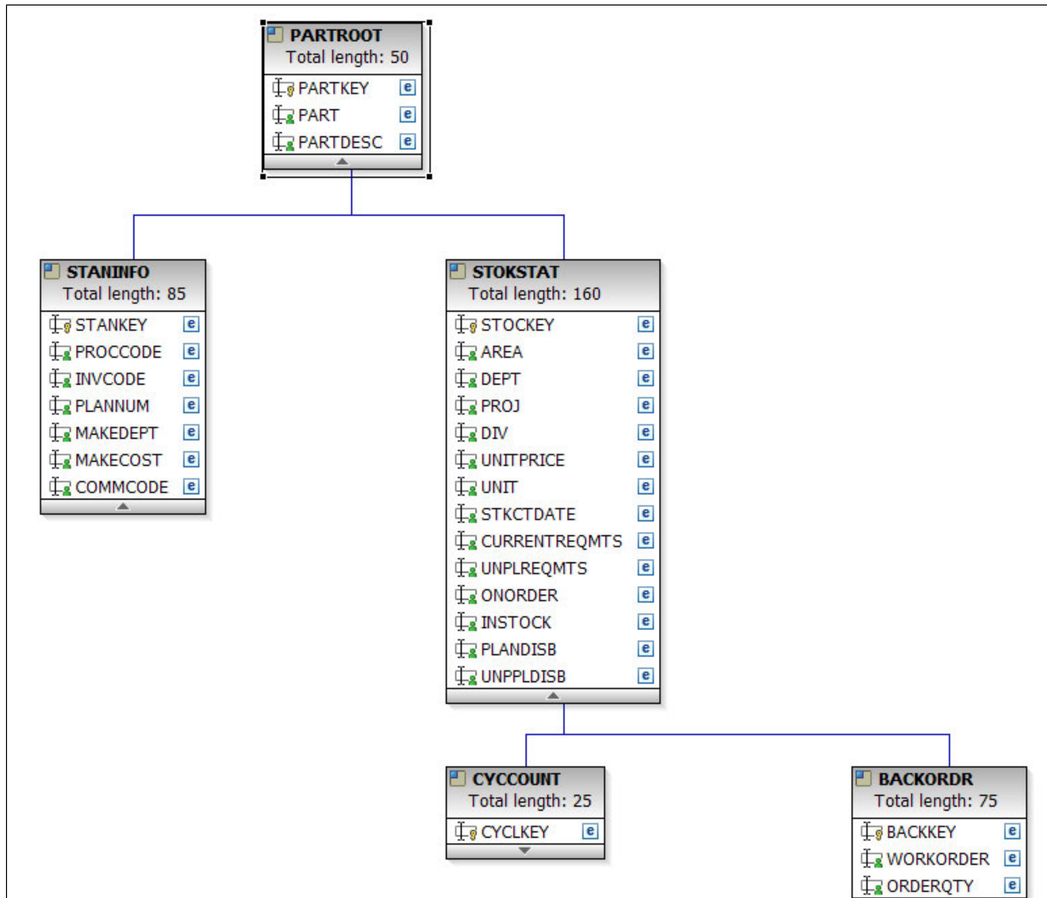


Figure 5-3 Structure of the sample part database

Only the IMS DB key fields were defined in the DBDs, so copybooks were imported to create a complete mapping that also contains the application defined database fields.

It is assumed that there is a working copy of either Rational Developer for System z, or Eclipse with IMS Enterprise Suite Explorer is installed. For information about how to install and use the plug-in, refer to *IMS 11 Open Database*, SG24-7856.

When the process of the mapping with the IMS Enterprise Suite Explorer is finished, a Java source file called the `DLIDatabaseView` is created. It contains the metadata for any Java based access to an IMS database. This DLIDatabaseview class later is required to be in the class path of the IMS Java JDBC driver as well as in the connection URL for use with the IMS Universal JDBC Driver for either Type 2 or Type 4 JDBC access to IMS data.

As of the writing of this book, IMS V12 provides only the ability to store IMS metadata in a catalog. It is not possible to use the IMS catalog as the metadata source for the IMS Universal JDBC driver.

The best process to map the IMS DB to an object hierarchy is when the current IMS DB is used as the source to create the mapping of the Java objects. This process includes the following steps:

1. Create the IMS DB metadata Java source class, DLIDatabaseView, using the IMS Enterprise Suite Explorer.

   The keys need to be in separate Java objects and it inherits the parent level keys in order to preserve the hierarchical levels for well performing access to IMS data. Hibernate then

builds WHERE clauses for SQL that contain the higher-level keys in the WHERE clauses and avoids IMS database scans when the higher-level keys are known.

2. Write down the key field names, data types, and lengths for all segments and their hierarchical structure. This can be obtained from the generated Java code DLI Database.

   The DLIDatabaseView generated Java code for the PART and STOKSTAT segments is shown in Example 5-20.

Example 5-20 DLIDatabaseView generated Java code for PART and STOKSTAT segments

```
// The following describes Segment: PARTROOT ("PARTROOT") in PCB: PCB01 ("PCB01")
    static DLITypeInfo[] PCB01PARTROOTArray= {
        new DLITypeInfo("PARTKEY", DLITypeInfo.CHAR, 1, 17, "PARTKEY",
DLITypeInfo.UNIQUE_KEY),
        new DLITypeInfo("PART", DLITypeInfo.CHAR, 3, 15),
        new DLITypeInfo("PARTDESC", DLITypeInfo.CHAR, 27, 20)
    };
    static DLISegment PCB01PARTROOTSegment= new DLISegment
        ("PARTROOT","PARTROOT",PCB01PARTROOTArray,50);

// The following describes Segment: STOKSTAT ("STOKSTAT") in PCB: PCB01 ("PCB01")
    static DLITypeInfo[] PCB01STOKSTATArray= {
        new DLITypeInfo("STOCKEY", DLITypeInfo.CHAR, 1, 16, "STOCKEY",
DLITypeInfo.UNIQUE_KEY),
        new DLITypeInfo("AREA", DLITypeInfo.CHAR, 3, 1),
        new DLITypeInfo("DEPT", DLITypeInfo.CHAR, 4, 2),
        new DLITypeInfo("PROJ", DLITypeInfo.CHAR, 6, 3),
        new DLITypeInfo("DIV", DLITypeInfo.CHAR, 9, 2),
        new DLITypeInfo("UNITPRICE", "S99999V999", DLITypeInfo.ZONEDDECIMAL, 21, 8),
        new DLITypeInfo("UNIT", DLITypeInfo.CHAR, 35, 4),
        new DLITypeInfo("STKCTDATE", DLITypeInfo.CHAR, 72, 3),
        new DLITypeInfo("CURRENTREQMTS", "S9999999", DLITypeInfo.ZONEDDECIMAL, 90, 7),
        new DLITypeInfo("UNPLREQMTS", "S9999999", DLITypeInfo.ZONEDDECIMAL, 98, 7),
        new DLITypeInfo("ONORDER", "S9999999", DLITypeInfo.ZONEDDECIMAL, 106, 7),
        new DLITypeInfo("INSTOCK", "S9999999", DLITypeInfo.ZONEDDECIMAL, 114, 7),
        new DLITypeInfo("PLANDISB", "S9999999", DLITypeInfo.ZONEDDECIMAL, 122, 7),
        new DLITypeInfo("UNPLDISB", "S9999999", DLITypeInfo.ZONEDDECIMAL, 130, 7)
    };
    static DLISegment PCB01STOKSTATSegment= new DLISegment
        ("STOKSTAT","STOKSTAT",PCB01STOKSTATArray,160);
```

The data required for the PART and STOKSTAT segments is listed in Table 5-1.

Table 5-1 Metadata required to create the Java objects for the keys and segment hierarchy

Parent segment	Segment name	Key name	Key data type	Key length
None	PARTROOT	PARTKEY	java.lang.String	17 bytes
PARTROOT	STOKSTAT	STOCKEY	java.lang.String	16 bytes
PARTROOT	STANINFO	STANKEY	java.lang.String	2 bytes
STOKSTAT	CYCCOUNT	CYCLKEY	java.lang.String	2 bytes
STOKSTAT	BACKORDR	BACKKEY	java.lang.String	10 bytes

3. Create the Java object for the key of the root segment:

   – Class is named PartKey with only one data item partKey of Java type String

- The serialVersionUID is required to avoid Java warnings
- The getter and setter method for the data item partKey with an optional self implemented toString method

The Java object class PartKey that reflects the key of the root segment named partKey is shown in Example 5-21.

Example 5-21 Java object class PartKey

```
public class PartKey {
   private static final long serialVersionUID = 1L;
   private String partKey;

   public PartKey(){
   }

   public String getPartKey() {
      return partKey;
   }

   public void setPartKey(String partKey) {
      this.partKey = partKey;
   }

   public String toString() {
      return "Partkey: "+getPartKey();
   }
}
```

> **Tip:** All Eclipse-based development environments have the ability to generate the getter and setter methods for a given Java data item. Right-click the data item and select the menu **Source → Generate getters and setters...** The wizard allows you to select multiple data items as well as where to insert the code.

4. Create the Java object for the key of the STOKSTAT segment.

   The Class is named *StocKey* and it inherits PartKey. Hibernate requires that it is serializable because StocKey has only one data item. The serialVersionUID is required to avoid Java warnings. Use the getter and setter method for the data item StocKey with an optional self implemented toString method. The Java object class StocKey is shown in Example 5-22. It shows the key of the STOKSTAT segment named StocKey as well as the hierarchical parentage to the PART segment by inheriting the PartKey class.

Example 5-22 Java object class StocKey

```
import java.io.Serializable;

public class StocKey extends PartKey implements Serializable{

   private static final long serialVersionUID = 1L;
   private String stocKey;

   public StocKey(){
   }

   public void setStocKey(String stanKey) {
      this.stocKey = stanKey;
   }
```

```
   public String getStocKey() {
      return stocKey;
   }

   public String toString() {
      return "Stockkey: "+getStocKey();
   }
}
```

5. Create the Java object for the key of the STANINFO segment.

   The Class is named *StanKey* that inherits PartKey and it is serializable as required by Hibernate. It has only one data item stanKey (the serialVersionUID is required to avoid Java warnings) as well as the getter and setter method for the data item stanKey with an optional self implemented toString method.

6. Create the Java object for the key of the CYCCOUNT segment.

   Class is named *CyclKey* that inherits StocKey (serializable is required by Hibernate) with only one data item cyclKey (the serialVersionUID is required to avoid Java warnings) as well as the getter and setter method for the data item cyclKey with an optional self implemented toString method.

7. Create the Java object for the key of the BACKORDR segment.

   Class is named *BackKey* that inherits StocKey (serializable is required by Hibernate) with only one data item backKey (the serialVersionUID is required to avoid Java warnings) as well as the getter and setter method for the data item backKey with an optional self implemented toString method.

   The key objects have been created successfully.

8. From the DLIDatabaseView (see Example 5-20 on page 86), it is required to make a list of the data items to create the mapping objects for the complete segments.

   The list of data items in Table 5-2 are also required to create the Hibernate mapping file.

Table 5-2 List of data items for segment PARTROOT

Data item name	Data item type	Data item length
partKey	PartKey (created earlier)	-
part	java.lang.String	15 bytes
partDesc	java.lang.String	20 bytes
stokStat	StokStat (to be created)	-
stanInfo	StanInfo (to be created)	-

   The PARTROOT segment has two child segments STANINFO and STOKSTAT. To have a relationship to the lower-level segments, Java Sets are used to reflect a possible 1:n relationship between PARTROOT and its child segments.

   Arrays and maps can only be used with segments that have integer type keys.

9. Create the Java object for the PARTROOT segment.

   Class is named *Part* and contains the key and data items for the segments as well as its getter and setter methods plus the two Set data items reflecting the 1:n relationships to the two child segments STANINFO and STOKSTAT with an optional self implemented toString method, as shown in Example 5-23 on page 89.

Example 5-23 Java object class part reflecting the PARTROOT segment

```
import java.util.Set;

public class Part {
   private static final long serialVersionUID = 1L;
   private String partKey;
   private String part;
   private String partDesc;

   //Array and Map should be used with Integer keys only
   private Set<?> stokStat;
   private Set<?> stanInfo;

   public Part(){
   }

   public String getPartKey() {
      return partKey;
   }

   public void setPartKey(String partKey) {
      this.partKey = partKey;
   }

   public String getPart() {
      return part;
   }

   public void setPart(String part) {
      this.part = part;
   }

   public String getPartDesc() {
      return partDesc;
   }

   public void setPartDesc(String partDesc) {
      this.partDesc = partDesc;
   }

   public void setStokStat(Set<?> stokStat) {
      this.stokStat = stokStat;
   }

   public Set<?> getStokStat() {
      return stokStat;
   }

   public void setStanInfo(Set<?> stanInfo) {
      this.stanInfo = stanInfo;
   }

   public Set<?> getStanInfo() {
      return stanInfo;
   }
```

```
   public String toString() {
      return "Partkey: "+getPartKey()+" Part: "+getPart()+" PartDesc: 
"+getPartDesc();
   }
}
```

10. Create the mapping Java class for the STANINFO segment.

    From the DLIDatabaseView (refer to Example 5-20 on page 86), it is required to make a list of the data items to create the mapping objects for the complete segments.

    The list of data items in Table 5-3 is also required to create the Hibernate mapping file.

Table 5-3 List of data items for the STOKSTAT segment

Data item name	Data item type	Data item length
stocKey	StocKey (created earlier)	-
area	java.lang.String	1 byte
dept	java.lang.String	2 bytes
proj	java.lang.String	3 bytes
div	java.lang.String	2 bytes
unitPrice	java.math.BigDecimal	8 bytes
unit	java.lang.String	4 bytes
stkctDate	java.lang.String	3 bytes
currentReqmts	java.math.BigDecimal	7 bytes
unplReqmts	java.math.BigDecimal	7 bytes
onOrder	java.math.BigDecimal	7 bytes
inStock	java.math.BigDecimal	7 bytes
planDisb	java.math.BigDecimal	7 bytes
unplDisb	java.math.BigDecimal	7 bytes
cyclCount	CyclCount (to be created)	-
backOrdr	BackOrdr (to be created)	-

The STOKSTAT segment has two child segments: CYCCOUNT and BACKORDR.

To have a relationship to the lower-level segments, Java Sets are used to reflect a possible 1:n relationship between STOKSTAT and its child. Arrays and Maps can only be used with segments that have Integer type keys. In addition to reflect the n:1 relationship for each STOKSTAT segment to its parent PARTROOT, a single data item part also needs to be added to the STOKSTAT Java class object.

11. Create the Java object for the STOKSTAT segment.

    Class is named StokStat and contains the key and data items for the segments as well as its getter and setter methods plus the part data item to point to the parent PARTROOT segment and the two Set<?> data items reflecting the 1:n relationships to the two child segments STANINFO and STOKSTAT with an optional self implemented toString method. See Example 5-24 on page 91, which has been shortened for readability.

Example 5-24 Java object class part reflecting the StokStat segment

```
import java.math.BigDecimal;
import java.util.Set;

public class StokStat {
   private StocKey stocKey;
   private String area;
   private String dept;
   private String proj;
   private String div;
   private BigDecimal unitPrice;
   private String unit;
   private String stkctDate;
   private BigDecimal currentReqmts;
   private BigDecimal unplReqmts;
   private BigDecimal onOrder;
   private BigDecimal inStock;
   private BigDecimal planDisb;
   private BigDecimal unplDisb;

   private Set<?> cyclCount;
   private Set<?> backOrdr;

   private Part part;

   public StokStat(){
   }

   public void setStocKey(StocKey stocKey) {
      this.stocKey = stocKey;
   }

   public StocKey getStocKey() {
      return stocKey;
   }

...
```

12. Create the Java class file mappings for all other segments, if required.

> **Note:** It is possible to work with a subset of the IMS DB segments and it might not be required to map all the segments against Java objects. If a segment between two hierarchical levels (for example, STOKSTAT) is omitted for access to the PARTROOT and CYCCOUNT segments, data access might be limited. For inserts and updates the key of the higher-level segment is required, but it is not possible if the STOKSTAT segment is omitted.
>
> One example is to map PARTROOT and STANINFO segments and access them because all others are not required from a business perspective. Another example is to only map PARTROOT, STOKSTAT, and CYCCOUNT, because they are in the same hierarchical path.
>
> When the path from the root segment to any lower-level segment is completely mapped, there are no restrictions to Select, Update, or Insert processing.

13. Create the Hibernate mapping when all the mappings are done. This can be done by annotations, which requires the annotation components of Hibernate to be in the class path, or to achieve XML-based (`.hbm.xml` file) mapping, which is the easier approach.

   To create the Hibernate mapping requires the information that is listed in Table 5-1 on page 86 and in Table 5-2 on page 88. The key definitions are the ID for the root segment and the composite IDs. The 1:n relationships for the child or dependent segments are implemented as sets with concatenated keys (except for the root segment) and one-to-many relationship type.

   The on-delete noaction parameter reflects the fact that when a segment is deleted, all its dependent segments are automatically deleted as well or deleted on cascade. The same applies that if the key of a segment is changed, all the dependent segments will automatically have that higher-level key changed.

   In IMS, there is no need to change the reference of the key in the dependent segments because it is implemented as a foreign key field that is automatically derived from the parent segments.

   The complete class mapping for the PARTROOT segment according to the Part Java class created earlier is displayed in Example 5-25.

Example 5-25 Class tag representing the Hibernate mapping for the PARTROOT segment

```
<class name="com.ibm.ims.hibernate.sample.Part" table="PARTROOT">
    <id name="partKey" column="PARTKEY"  type="java.lang.String" length="17">
       <generator class="assigned"/>
    </id>
    <property name="part" column="PART" type="java.lang.String" length="15"/>
    <property name="partDesc" column="PARTDESC"  type="java.lang.String"
length="20"/>
    <set name="stokStat" cascade="none">
        <key column="PARTROOT_PARTKEY" on-delete="noaction" update="false"/>
        <one-to-many class="com.ibm.ims.hibernate.sample.StokStat"/>
    </set>
    <set name="stanInfo" cascade="none">
        <key column="PARTROOT_PARTKEY" on-delete="noaction" update="false"/>
        <one-to-many class="com.ibm.ims.hibernate.sample.StanInfo"/>
    </set>
 </class>
```

   The lower-level segments have composite IDs and concatenated keys. They use the foreign key definitions that come automatically from the IMS JDBC Driver and the name is concatenated SEGMENT_KEYFIELD. The n:1 relationship to the parent segment is implemented as a many-to-one mapping for Hibernate.

   The complete class mapping for the STOKSTAT segment according to the StokStat Java class created earlier is shown in Example 5-26.

Example 5-26 Class tag representing the Hibernate mapping for the STOKSTAT segment

```
<class name="com.ibm.ims.hibernate.sample.StokStat" table="STOKSTAT">
    <composite-id name="stocKey">
        <key-property name="partKey" column="PARTROOT_PARTKEY"
type="java.lang.String" length="17"/>
        <key-property name="stocKey" column="STOCKEY" type="java.lang.String"
length="10"/>
        <generator class="assigned"/>
    </composite-id>
    <property name="area" column="AREA"  type="java.lang.String" length="1"/>
```

```
    <property name="dept" column="DEPT"  type="java.lang.String" length="2"/>
    <property name="proj" column="PROJ"  type="java.lang.String" length="3"/>
    <property name="div" column="DIV"  type="java.lang.String" length="2"/>
    <property name="unit" column="UNIT"  type="java.lang.String" length="4"/>
    <property name="stkctDate" column="stkctDate"  type="java.lang.String"
length="3"/>
    <set name="cyclCount" cascade="none">
        <key>
          <column name="PARTROOT_PARTKEY" length="17"></column>
          <column name="STOKSTAT_STOCKEY" length="16"></column>
     </key>
        <one-to-many class="com.ibm.ims.hibernate.sample.CyclCount"/>
    </set>
    <set name="backOrdr" cascade="none">
        <key>
          <column name="PARTROOT_PARTKEY" length="17"></column>
          <column name="STOKSTAT_STOCKEY" length="16"></column>
        </key>
        <one-to-many class="com.ibm.ims.hibernate.sample.BackOrdr"/>
    </set>
   <many-to-one name="part" class="com.ibm.ims.hibernate.sample.Part"
insert="false" update="false" cascade="none">
      <column name="PARTROOT_PARTKEY" length="17"></column>
   </many-to-one>
 </class>
```

The generator class for the keys is assigned because the IMS database does not support automatic key generation like auto increment and in order to add a new non-root segment, the parentage to one or more parent segments needs to be established for the insert to work.

For example, to add a STOKSTAT segment using Hibernate, the key of the PARTROOT segment where the STOKSTAT segment should be placed needs to be specified. This is also the reason that composite IDs and concatenated keys were used for the Hibernate key mapping of all non-root segments, such that the specification of all keys in the hierarchic sequence up to the root segment is enforced by the Hibernate layer.

Note: The complete Hibernate mapping is part of the sample, which is part of the Rational Developer for System z workspace that contains the IMS samples. Since the workspace is a compressed file, it should be possible to extract it and import it into other Eclipse and non-Eclipse-based development environments.

To create the Hibernate configuration consists of some naming conventions and the configuration of the IMS Universal JDBC driver to be used for accessing the IMS database. See Example 5-27.

Example 5-27 IMS Universal JDBC driver remote usage configuration as part of the hibernate.cfg.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
```

```
 <property
name="connection.url">jdbc:ims://zserveros.dfw.ibm.com:7001/class://samples.ivp.op
endb.DFSSAM09DatabaseView:user=USERID;password=PASSWORD;</property>
   <property name="connection.driver_class">com.ibm.ims.jdbc.IMSDriver</property>
   <property name="dialect">org.hibernate.dialect.DB2Dialect</property>
 <property
name="transaction.factory_class">org.hibernate.transaction.JDBCTransactionFactory<
/property>
    <!--  thread is the short name for
      org.hibernate.context.ThreadLocalSessionContext
      and let Hibernate bind the session automatically to the thread
    -->
    <property name="current_session_context_class">thread</property>
    <!-- this will show us all sql statements -->
    <property name="hibernate.show_sql">true</property>
   <!-- mapping files -->
   <mapping resource="com/ibm/ims/hibernate/sample/part.hbm.xml" />
</session-factory>
</hibernate-configuration>
```

Hibernate requires log4j properties and depending on the Hibernate version or function level (annotations require additional classes), a certain number of JAR files are required in the class path. The JAR files come either directly from the Hibernate website or are references to other open source projects, such as Apache Commons and Log4J.

For the Hibernate tests that do not use caching, the following JAR files are required to be in the class path:

- `antldr-2.7.6.jar`
- `commons-collections-3.1.jar`
- `dom4j-1.6.1.jar`
- `hibernate3.jar, imsudb.jar`
- `javaassist-3.9.0.jar`
- `jta-1.1.jar`
- `log4j-1.2.15.jar`
- `slf4j-api-1.5.11.jar`
- `slf4j-log4j12-1.5.11.jar`
- `slf4j-log4j12-1.5.11.jar`

An application based on the Hibernate API needs to be written to be able to load an object based on the data in the IMS DB. The easiest way is to encapsulate the Hibernate access using, for example *getObject method*, and have a simple application; see Example 5-28.

Example 5-28 getObject method implementation to hide the Hibernate access from the application

```
//method to load a segment according to the key specified
  private static Object getObject(Class<?> clazz, Serializable id) {
    Transaction tx = null;
    Session session = SessionFactoryUtil.getInstance().getCurrentSession();
    Object object = null;
    try {
      tx = session.beginTransaction();
      object = session.get(clazz, id);
      if (object  == null) {
      System.out.println("Object loaded from Session.");
      object = session.load(clazz, id);
```

```
      } else {
      System.out.println("Object loaded from Database");
      }
      tx.commit();
      return object;
    } catch (RuntimeException e) {
      if (tx != null && tx.isActive()) {
        try {
// Second try catch as the rollback could fail as well
          tx.rollback();
        } catch (HibernateException e1) {
          logger.debug("Error rolling back transaction");
        }
        e.printStackTrace();
// throw again the first exception
        throw e;
      }
    }
    return null;
  }
```

The sample code in Example 5-29 is used to access the Part Object with key 02AN960C10.

Example 5-29 Load a Part Object from the PARTROOT segment using the getObject method

```
   Part testpart = (Part)getObject(Part.class, new String("02AN960C10"));
   System.out.println(testpart.getPartKey());
```

Example 5-30 depicts the createObject method implemented to create a new object.

Example 5-30 createObject method to hide the Hibernate access from the application

```
//method to create a segment according to the object specified
  private static void createObject(Object object) {
    Transaction tx = null;
    Session session = SessionFactoryUtil.getInstance().getCurrentSession();
    try {
      tx = session.beginTransaction();
      session.save(object);
      tx.commit();
    } catch (RuntimeException e) {
      if (tx != null && tx.isActive()) {
        try {
// Second try catch as the rollback could fail as well
          tx.rollback();
        } catch (HibernateException e1) {
          logger.debug("Error rolling back transaction");
        }
// throw again the first exception
        throw e;
      }
    }
  }
```

The sample code for creating a new object is shown in Example 5-31.

Example 5-31 Create a Part Object using the createObject method

```
//create a new part
    Part denisPart = new Part();
    denisPart.setPartKey("02This is partXX");
    denisPart.setPart("This is partXX");
    denisPart.setPartDesc("very sweet");
    denisPart.setStokStat(null);
    //make persistent if its not there
        System.out.println(denisPart.getPartKey());
      try {
    createObject(denisPart);
    } catch (Exception e) {
    System.out.println("Guess the part already exists.");
    e.printStackTrace();
    }
    // our instance has a primary key now:
    logger.debug("{}", denisPart);
```

The updateObject method implementation to update an object is shown in Example 5-32.

Example 5-32 updateObject method implementation to hide the Hibernate access from the application

```
//method to update a segment according to the object specified
  private static void updateObject(Object object) {
    Transaction tx = null;
    Session session = SessionFactoryUtil.getInstance().getCurrentSession();
    try {
      tx = session.beginTransaction();
      session.update(object);
      tx.commit();
    } catch (RuntimeException e) {
      if (tx != null && tx.isActive()) {
        try {
// Second try catch as the rollback could fail as well
          tx.rollback();
        } catch (HibernateException e1) {
          logger.debug("Error rolling back transaction");
        }
// throw again the first exception
        throw e;
      }
    }
  }
```

Example 5-33 shows the sample code to update an object.

Example 5-33 Update a Part Object using the updateObject method

```
//Change a value of the object
    System.out.println(denisPart.getPartKey());
    denisPart.setPartDesc("Norther Forest Honey");
    //make persistent
    updateObject(denisPart);
```

```
    System.out.println(denisPart.getPartKey());
```

The method implementation to delete an object is shown in Example 5-34.

Example 5-34 deleteObject method implementation to hide the Hibernate access from the application

```
//method to delete a segment according to the object specified
  private static void deleteObject(Object object) {
    Transaction tx = null;
    Session session = SessionFactoryUtil.getInstance().getCurrentSession();
    try {
      tx = session.beginTransaction();
      session.delete(object);
      tx.commit();
    } catch (RuntimeException e) {
      if (tx != null && tx.isActive()) {
        try {
// Second try catch as the rollback could fail as well
          tx.rollback();
        } catch (HibernateException e1) {
          logger.debug("Error rolling back transaction");
        }
// throw again the first exception
        throw e;
      }
    }
  }
```

The sample code to delete an object is shown in Example 5-35.

Example 5-35 Delete a Part Object using the deleteObject method

```
//delete the Part
    deleteObject(denisPart);
```

Note: Saving or updating an object in a Hibernate session does not mean that it is immediately persisted into the database. This can happen later. Therefore, in any batch that uses Hibernate, it is required to have a session.flush() as the last statement. Running the preceding example without session.flush() did not always result in writing changes to the IMS DB.

Example 5-36 shows the Hibernate configuration that is required to run with IMS.

Example 5-36 Hibernate properties used for running in IMS (hibernate.cfg.xml)

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
   <property name="connection.url"
>jdbc:ims:class://samples.ivp.opendb.DFSSAM09DatabaseView</property>
   <property name="connection.driver_class">com.ibm.ims.jdbc.IMSDriver</property>
   <property name="dialect">org.hibernate.dialect.DB2Dialect</property>
 <property name="hibernate.connection.pool_size">0</property>
```

```
   <property
name="transaction.factory_class">org.hibernate.transaction.JTATransactionFactory</
property>
    <property name="current_session_context_class">jta</property>
    <property
name="hibernate.transaction.manager_lookup_class">org.hibernate.transaction.IMSTra
nsactionManagerLookup</property>
    <!-- this will show us all sql statements -->
    <property name="hibernate.show_sql">yes</property>
   <!-- mapping files -->
   <mapping resource="com/ibm/ims/hibernate/sample/part.hbm.xml" />
</session-factory>
</hibernate-configuration>
```

The IMSTransactionManagerLookup class points to the Java Naming and Directory Interface (JNDI) names that are provided by the container. IMS does currently not provide JNDI services (there is a possible future function, that limited support as IMSNamingFactory and IMSNamingContextImpl will be provided). Therefore, it is required to store the name of the IMS implementation for UserTransactionManager and UserTransaction in the JNDI namespace that Hibernate loads from the class in the hibernate.transaction.manager_lookup_class property.

In the sample Hibernate properties shown in Example 5-36 on page 97, the simple javaURLContextFactory provider was used, which is basically a piece of code that uses a flat file as storage.

However, currently it is required to store the JNDI properties before entering a Hibernate-based IMS application. The Java code to store the JNDI properties in this case looks like the display in Example 5-37. This currently is done in the user application.

Example 5-37 Sample Java code to store JNDI properties with the javaURLContextFactory provider

```
        IMSUserTransactionManager dummy = new IMSUserTransactionManager();
          System.setProperty(Context.INITIAL_CONTEXT_FACTORY,
"org.apache.naming.java.javaURLContextFactory");
          System.setProperty(Context.URL_PKG_PREFIXES, "org.apache.naming");
          InitialContext ctx = new InitialContext();
         ctx.createSubcontext("java:");
          ctx.createSubcontext("java:comp");
        ctx.bind("java:comp/UserTransactionManager", dummy);
        ctx.bind("java:comp/UserTransaction", dummy.getTransaction());
```

Now the Hibernate program can be executed. A sample workspace is provided as part of the downloads. There is a Java class with a main method, which can be easily executed. The PSB used for the preceding sample is DFSSAM09, which should have been generated and installed in the IMS system that is to be used.

5.3.4 Using JPA as OR mapper with DB2

To use JPA in IMS batch there is not much work required. A working sample that runs with a Type 4 JDBC Driver against an existing DB2 for z/OS installation can be used. The following items also need to be addressed:

- Switch to use JTATransaction as the transaction type.

- Ensure that all required classes are in the class path of the IMS Batch application configuration definitions.
- Change the Connection URL for the DB2 JDBC Driver to be compatible with the requirements of the DB2 Universal JDBC Driver running in an IMS dependent region. This applies to JMP, JBP, message processing program (MPP), or BMP.

Note: Any commit() calls are ignored because IMS manages the unit of work.

If in batch there is a requirement for consistency points, use the Java methods for issuing IMS checkpoint calls.

JPA sample

In this section we review a JPA sample:

1. For the JPA sample, the following JAR files are required to be in the project class path:
   - imsutm.jar for the IMS Transaction Manager services
   - openjpa-all-2.1.1.jar for the OpenJPA classes

The class representing Stock used for the JPA sample is shown in Example 5-38.

Example 5-38 Class Stock for the JPA sample

```
package model;

import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.SequenceGenerator;

import org.apache.openjpa.persistence.jdbc.VersionColumn;

@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
@VersionColumn
public class Stock {
   @Id
   @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="StockSeq")
   @SequenceGenerator(name="StockSeq", sequenceName="STOCK_SEQ")
   private long id;

   @Basic
   private double course;

   @Basic(fetch=FetchType.LAZY)
   private String name;

   @Basic
   private String descr;

   public String getDescr() {
```

```
      return descr;
   }

   public void setDescr(String descr) {
      this.descr = descr;
   }

   public Stock(){
      super();
   }

   public Stock(double course, String name){
      this.course = course;
      this.name = name;
   }

   public void setId(long id) {
      this.id = id;
   }

   public long getId() {
      return id;
   }

   public void setCourse(double course) {
      this.course = course;
   }

   public double getCourse() {
      return course;
   }

   public void setName(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }

   public String toString(){
      return "ID: " + id + " Course: " + course + " Name: " + name;
   }
}
```

Similar to Hibernate, the IMS Java classes do not set up the JNDI properties that are required to run JPA, so the JNDI setup needs to be done at the start of the program.

2. The sample code uses the javaURLContextFactory, as shown in Example 5-39.

Example 5-39 JNDI setup currently required for JPA in IMS Java enabled regions

```
        IMSUserTransactionManager dummy = new IMSUserTransactionManager();
          System.setProperty(Context.INITIAL_CONTEXT_FACTORY,
"org.apache.naming.java.javaURLContextFactory");
          System.setProperty(Context.URL_PKG_PREFIXES, "org.apache.naming");
          InitialContext ctx = new InitialContext();
```

```
        ctx.createSubcontext("java:");
          ctx.createSubcontext("java:comp");
        ctx.bind("java:comp/UserTransactionManager", dummy);
        ctx.bind("java:comp/UserTransaction", dummy.getTransaction());
```

3. Example 5-40 shows sample code used to persist random instances of Stock data.

Example 5-40 Sample code to persist random instances of Stock data

```
public static void fillDatabase()
    {
        EntityManager em = factory.createEntityManager();
        try
        {
        for(int i = 0; i < 5; i++)
        {
        Stock s = new Stock((new Random()).nextDouble() * 100D, (new
StringBuilder("STOCK")).append(i).toString());
        s.setDescr("d");
        em.persist(((Object) (s)));
        //em.lock(((Object) (s)), LockModeType.OPTIMISTIC);
        em.lock(((Object) (s)), LockModeType.PESSIMISTIC_READ);
        }
        em.flush();
        }
        catch(Exception e)
        {
            System.out.println("----------------EXCEPTION--------------");
            e.printStackTrace();
        }
        em.close();
    }
```

4. The persistence.xml is required to run JPA in IMS JVM enabled regions, as shown in Example 5-41.

Example 5-41 persistence.xml required for JPA in IMS Java enabled regions

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
   xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">


   <persistence-unit name="Stock" transaction-type="JTA">
      <provider>org.apache.openjpa.persistence.PersistenceProviderImpl</provider>
      <mapping-file>META-INF/orm.xml</mapping-file>
      <class>model.Stock</class>
      <class>model.StockDate</class>
      <class>model.Trader</class>
      <properties>
         <property name="openjpa.jdbc.DBDictionary"
value="org.apache.openjpa.jdbc.sql.DB2Dictionary"/>
         <property name="openjpa.ConnectionDriverName"
value="COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver" />
```

```
        <property name="openjpa.ConnectionURL"value="jdbc:db2os390sqlj:" />
        <property name="openjpa.TransactionMode" value="managed"/>
          <property name="openjpa.ConnectionFactoryMode" value="managed"/>
          <property name="openjpa.ManagedRuntime"
value="jndi(TransactionManagerName=java:comp/UserTransactionManager)"/>
          <property name="openjpa.jdbc.Schema" value="GAEBLER"/>
        <property name="openjpa.Log" value="DefaultLevel=TRACE" />
      </properties>

   </persistence-unit>
</persistence>
```

The connection URL and the JDBC Driver Class name are those that are supported in IMS JVM enabled environments as type 2 JDBC Connectivity. However, in the sample workspace there are comments with the Type 4 connection properties that were used to test the sample program locally. After a successful test, they are changed to Type 2, packaged to a JAR file, uploaded to z/OS, and executed as IMS BMP.

5. The orm.xml that is required to run JPA with DB2 in IMS JVM enabled regions is shown in Example 5-42.

Example 5-42 orm.xml required for JPA in IMS Java enabled regions

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings version="1.0" xmlns="http://java.sun.com/xml/ns/persistence/orm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
http://java.sun.com/xml/ns/persistence/orm_1_0.xsd">
<persistence-unit-metadata>
<persistence-unit-defaults>
<schema>GAEBLER</schema>
</persistence-unit-defaults>
</persistence-unit-metadata>
</entity-mappings>
```

The sample can now be executed in IMS JVM enabled environments.

5.4 Summary

This chapter has provided some samples for IMS Java batch program, samples for Java configuration, and IMS BMP calls and Java frameworks used with IMS Java.

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.

- *IMS 11 Open Database,* SG24-7856
- *IMS Connectivity in an On Demand Environment: A Practical Guide to IMS Connectivity*, SG24-6794
- *Using Integrated Data Management To Meet Service Level Objectives,* SG24-7769
- *DB2 9 for z/OS Stored Procedures: The CALL and Beyond*, SG24-7604

You can search for, view, download or order these documents and other IBM Redbooks, Redpapers, Web Docs, drafts, and additional materials, at the following website:

ibm.com/redbooks

Online resources

These websites are also relevant as further information sources:

- IMS homepage

  http://www.ibm.com/ims
- IMS Version 11 information

  http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp?topic=%2Fcom.ibm.ims11.doc%2Fimshome_v11.htm
- Debug Tool plug-in

  http://www-01.ibm.com/support/docview.wss?uid=swg24026610
- IMS/DC Execution Option

  http://pic.dhe.ibm.com/infocenter/wtxdoc/v8r3m0/topic/com.ibm.websphere.dtx.imsdc.doc/topics/g_imsdc_exec_Introduction.htm
- IBM Monitoring and Diagnostic Tools for Java - Health Center

  http://www.ibm.com/developerworks/java/jdk/tools/healthcenter
- Java Standard Editions website

  http://www-03.ibm.com/systems/z/os/zos/tools/java
- JVM technical articles

  http://www.oracle.com/technetwork/articles/java/index.html

© Copyright IBM Corp. 2014. All rights reserved.

- IBM HeapAnalyzer

  https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=4544bafe-c7a2-455f-9d43-eb866ea60091
- JConsole

  http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/index.jsp?topic=%2Fcom.ibm.java.aix.70.doc%2Fdiag%2Ftools%2FJConsole.html
- IBM Data Studio

  http://www-142.ibm.com/software/products/us/en/data-studio
- Optim Development Studio

  http://www-01.ibm.com/software/data/optim/development-studio
- IBM IMS Batch Terminal Simulator

  http://www-01.ibm.com/software/data/db2imstools/imstools/imsbts
- IBM Debug Tool Plug-in for Eclipse

  http://www-01.ibm.com/support/docview.wss?uid=swg24026610
- IBM Debug Tool Plug-in for Eclipse, instructions for V12

  ftp://public.dhe.ibm.com/software/htp/pdtools/plugins/DT_plugin_V12100_readme.pdf
- Debug Tool manuals

  http://www-01.ibm.com/software/awdtools/debugtool/library
- DB2 Analytics Accelerator for z/OS powered by Netezza technology

  http://www-01.ibm.com/software/data/db2/zos/analytics-accelerator
- WebSphere MQ for z/OS Information center

  http://pic.dhe.ibm.com/infocenter/wmqv7/v7r5/index.jsp
- *Enterprise COBOL for z/OS Programming Guide*

  http://pic.dhe.ibm.com/infocenter/pdthelp/v1r1/index.jsp?topic=%2Fcom.ibm.entcobol.doc_4.2%2FPGandLR%2Fref%2Frlpsinvo.htm
- *JZOS for z/OS SDKs Cookbook*

  https://www.ibm.com/services/forms/preLogin.do?source=zossdkcookbook
- Manuals for IBM 31-bit SDK for z/OS, Java Technology Edition, V6.0.1, and the SDK Guide

  http://www-03.ibm.com/systems/z/os/zos/tools/java/products/sdk601_31.html#j6content
- WebSphere Transformation Extender documentation

  http://pic.dhe.ibm.com/infocenter/wtxdoc/v8r3m0/index.jsp

Help from IBM

IBM Support and downloads

ibm.com/support

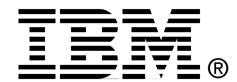
IBM Global Services

ibm.com/services

IBM Redbooks

New Ways of Running Batch Applications on z/OS: Volume 4 IBM IMS

(0.2"spine)
0.17"<->0.473"
90<->249 pages



New Ways of Running Batch Applications on z/OS

Volume 4 IBM IMS

Technology overview

Modernization options

Samples

Mainframe computers play a central role in the daily operations of many of the world's largest corporations. Batch processing is still a fundamental, mission-critical component of the workloads that run on the mainframe. A large portion of the workload on IBM z/OS systems is processed in batch mode.

This IBM Redbooks publication is the fourth volume in a series of four. They address new technologies introduced by IBM to facilitate the use of hybrid batch applications that combine the best aspects of Java and procedural programming languages such as COBOL. This volume focuses on the latest enhancements in IBM IMS batch support. IMS has been available to clients for 45 years as IMS Transaction Manager, IMS Database Manager, or both.

The audience for this book includes IT architects and application developers with a focus on batch processing on the z/OS platform.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-8119-00 ISBN 0738439398