

Java Stand-alone Applications on z/OS Volume II

Using JZOS to develop, deploy and run
Java applications

Exploring the Java SDK 5

Java Native Interface



Alex Louwe Kooijmans
Jonathan M Barney
Patrick Bruinsma
Venugopal Kailaikurthi
Christian Strauer
Kenichi Yoshimura



International Technical Support Organization

Java Stand-alone Applications on z/OS Volume II

December 2006

Archived

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

First Edition (December 2006)

This edition applies to Java™ Version 1.5.0, and Java™ 2 Runtime Environment, Standard Edition (build pmz31devifx-20060524 (SR2))

© Copyright International Business Machines Corporation 2006. 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 | vii |
| Trademarks | viii |
| Preface | ix |
| The team that wrote this redbook | ix |
| Become a published author | xi |
| Comments welcome | xi |
| Chapter 1. Introduction | 1 |
| 1.1 Why Java batch on z/OS? | 2 |
| 1.1.1 Major enhancements in SDK 5.0 | 2 |
| 1.1.2 Conclusion | 3 |
| Chapter 2. Java on z/OS - The basics | 5 |
| 2.1 Java overview | 6 |
| 2.1.1 Java on z/OS | 7 |
| 2.1.2 Considerations when using Java on z/OS | 10 |
| 2.2 Garbage collection | 13 |
| 2.2.1 Garbage Collection in IBM SDK 5.0 | 14 |
| 2.2.2 Detailed description of Garbage Collection | 15 |
| Chapter 3. Job management using JZOS | 25 |
| 3.1 Positioning of JZOS | 26 |
| 3.2 JZOS launcher settings and return codes | 28 |
| 3.3 Art of job management using JZOS | 30 |
| 3.3.1 Best practice considerations | 36 |
| 3.4 Console communication with batch jobs | 42 |
| 3.4.1 MVS console communication overview | 42 |
| 3.4.2 JZOS APIs for console communication | 44 |
| 3.4.3 JZOS console communication example | 46 |
| Chapter 4. Java development and job management with Eclipse | 49 |
| 4.1 Overview | 50 |
| 4.2 Setting up Eclipse for host interaction | 51 |
| 4.2.1 Prerequisites | 51 |
| 4.2.2 Set up Ant FTP support in Eclipse | 52 |
| 4.2.3 Import sample project | 52 |
| 4.2.4 Configure the project | 54 |
| 4.3 Developing Java applications | 60 |

| | |
|--|------------|
| 4.3.1 Submitting jobs and controlling output | 60 |
| 4.3.2 Customizing the Java application | 63 |
| 4.4 Debugging | 67 |
| 4.4.1 Setting options | 67 |
| 4.4.2 Working with the Eclipse debugger | 69 |
| Chapter 5. Using WebSphere Developer for zSeries | 71 |
| 5.1 Overview of WDz | 72 |
| 5.2 Setting up a basic connection with the host | 77 |
| 5.3 Editing MVS file systems with WDz | 78 |
| 5.4 Managing MVS jobs with WDz | 80 |
| 5.5 Remote debugging using WDz | 81 |
| 5.6 Remote debugging of a Java application with JNI calls | 83 |
| Chapter 6. Java Native Interface (JNI) | 85 |
| 6.1 Introducing the Java Native Interface (JNI) | 86 |
| 6.1.1 Basic elements of JNI | 88 |
| 6.1.2 z/OS 64-bit considerations | 92 |
| 6.1.3 JNI and the Garbage Collector | 93 |
| 6.2 Getting started with a JNI application | 93 |
| 6.2.1 Developing a simple JNI program | 94 |
| 6.3 More advanced JNI usage | 97 |
| 6.3.1 Calling system services from a Java application | 97 |
| 6.3.2 Calling a COBOL program from a Java application | 101 |
| 6.4 Integrating Java and COBOL using IBM Enterprise Object-Oriented COBOL | 105 |
| 6.4.1 IBM Enterprise COBOL for z/OS introduction | 106 |
| 6.4.2 Preparing object-oriented applications under UNIX | 106 |
| 6.4.3 Java program calling COBOL | 108 |
| 6.4.4 COBOL program calling Java | 109 |
| Chapter 7. I/O using the JZOS toolkit API | 111 |
| 7.1 JZOS toolkit library introduction | 112 |
| 7.2 JZOS classes | 113 |
| 7.3 Usage examples | 114 |
| 7.3.1 Copying members of a data set using JZOS FileFactory | 114 |
| 7.3.2 Copying members of a data set using ZFile | 117 |
| 7.3.3 BPXWDYN | 118 |
| 7.3.4 Working with VSAM files | 120 |
| 7.4 JZOS versus JRIO | 126 |
| Chapter 8. Java problem determination | 129 |
| 8.1 Typical Java problems | 130 |
| 8.1.1 OutOfMemory problems | 130 |

| | |
|---|------------|
| 8.1.2 Hang or deadlock problems | 131 |
| 8.1.3 Crash problems..... | 132 |
| 8.1.4 High CPU or performance problems..... | 135 |
| 8.2 Diagnostic Tooling Framework for Java (DTFJ) | 135 |
| 8.3 Guidelines - using Java and tuning options..... | 136 |
| 8.3.1 Guidelines for using 31-bit versus 64-bit Java | 136 |
| 8.3.2 Java5 tuning options | 136 |
| Chapter 9. Java Virtual Machine Tool Interface | 139 |
| 9.1 Java Virtual Machine Tool Interface (JVMTI) | 140 |
| 9.2 Simple example to demonstrate usage of JVMTI | 140 |
| 9.3 Transition from JVMPI to JVMTI | 144 |
| Appendix A. Additional material | 147 |
| Locating the Web material | 147 |
| Using the Web material | 147 |
| How to use the Web material | 148 |
| Related publications | 149 |
| IBM Redbooks | 149 |
| Other publications | 149 |
| Online resources | 150 |
| How to get IBM Redbooks | 150 |
| Help from IBM | 150 |
| Index | 151 |

Notices

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

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

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

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 Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites 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.

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.

Trademarks

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

| | | |
|-----------------------|---|------------|
| AIX® | MQSeries® | RMF™ |
| alphaWorks® | MVS™ | System z™ |
| CICS® | OS/390® | System z9™ |
| DB2® | OS/400® | VisualAge® |
| i5/OS® | PR/SM™ | WebSphere® |
| IBM® | Rational® | z/OS® |
| IMS™ | Redbooks™ | zSeries® |
| IMS/ESA® | Redbooks (logo)  ™ | z9™ |
| Language Environment® | RACF® | |

The following terms are trademarks of other companies:

NOW, and the Network Appliance logo are trademarks or registered trademarks of Network Appliance, Inc. in the U.S. and other countries.

Java, JDBC, JDK, JVM, J2EE, Sun, Sun Microsystems, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Pentium, Intel logo, Intel Inside logo, and Intel Centrino logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Preface

This IBM® Redbook is the second redbook in a series of two about Java™ stand-alone applications on z/OS®. We recommend using this document as a complement to *Java stand-alone Applications on z/OS Volume I*, SG24-7177.

This book explains some topics discussed in Volume 1 in more depth, and also provides information about additional topics. It pays special attention to JZOS, a newly integrated function in the z/OS SDKs that can be used to run Java batch jobs and perform I/O.

This document also features other interesting topics, such as the use of Eclipse and WebSphere® Developer for zSeries® for integrated development of stand-alone Java applications, Java Native Interface (JNI), the new Java Virtual Machine Tool Interface (JVMTI) and problem determination with SDK 5 on z/OS.

The team that wrote this redbook

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

Alex Louwe Kooijmans is a Project Leader with the International Technical Support Organization (ITSO), Poughkeepsie Center, and specializes in WebSphere, Java and SOA on System z™ with a focus on integration, security, high availability and application development. Previously he worked as a Client IT Architect in the Financial Services sector with IBM in The Netherlands, advising financial services companies on IT issues such as software and hardware strategy and on demand. Alex has also worked at the Technical Marketing Competence Center for zSeries and Linux® in Boeblingen, Germany, providing support to customers starting up with Java and WebSphere on zSeries. From 1997 to 2002, Alex completed a previous assignment with the ITSO, managing various IBM Redbooks™ projects and delivering workshops around the world.

Jonathan M Barney is a Staff Software Engineer in USA. He has six years of experience in z/OS and Java and USS. He holds a B.S. degree in Computer Science from Clarkson University. Jonathan's areas of expertise include Java, RACF® Java Security and WebSphere.

Patrick Bruinsma is a Senior IT Specialist working for Software Group Services in the Netherlands. He has almost nine years of experience in mainframe environments on OS/390® and z/OS platforms. His areas of expertise include DB2®, MQSeries®, WebSphere MQ Workflow, Blaze Advisor, CICS®, WebSphere Application Server and z/OS UNIX®. Patrick has written extensively and teaches IBM classes on various aspects of z/OS UNIX.

Venugopal Kailaikurthi is a Staff Software Engineer with the Java Technology Center at Bangalore, India. He has six years of experience in Java Virtual Machine Technology. He holds a Bachelor's degree in Electronics from Gulbarga University. Venugopal's areas of expertise include C, C++, Java and UNIX. He has written articles on Java and contributed to the Java Diagnostic Guide.

Christian Strauer is an IT Specialist in the Systems and Technology Group in Germany. He has almost four years of experience in Java development and has focused on z/OS Java stand-alone applications for a year. Christian holds a diploma in Applied Computer Sciences from the University of Cooperative Education in Mannheim. His areas of expertise are Java and SOA on z/OS.

Kenichi Yoshimura is a Software Engineer in the Australia Development Laboratory based in Perth, Australia. Since joining IBM in 2004, he has worked on several System z software products including IBM Fault Analyzer for z/OS and IBM SCLM Developer Toolkit. Kenichi's main areas of expertise are Java on z/OS and Eclipse plug-in development. He holds a Master's Degree in Engineering Science from the University of Melbourne.

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

Richard Conway and Robert Haimowitz
International Technical Support Organization, Poughkeepsie Center

Norman Aaronson, Brian Beegle, Clarence Clark, Donald Durand, Clark Goodrich, Bob St. John, John Rankin
IBM Poughkeepsie lab

Larry England
IBM San Jose

Cynthia Krauss (Cindy)
IBM Raleigh (RTP)

Steve Goetze and Kirk Wolf
Dovetailed Technologies, LLC

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

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 Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- ▶ Use the online **Contact us** review redbook form found at:

ibm.com/redbooks

- ▶ Send your comments in an email to:

redbook@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

Introduction

The purpose of this redbook is two-fold:

- ▶ Introduce additional topics related to using Java in stand-alone or batch programs.
- ▶ Discuss major new enhancements in SDK 5.0.

1.1 Why Java batch on z/OS?

There are many reasons for using Java in a batch or stand-alone mode on z/OS. The Java language can be used like any other programming language in a batch job, shell script, or from the command line, but the Java language is in most cases more powerful than any other language. Let's give some examples of what you can do with Java on z/OS in a batch environment:

- ▶ Access relational databases and call stored procedures using the JDBC™ or SQLJ APIs
- ▶ Read from and write to various types of input files, such as UNIX files and MVS™ data sets including VSAM
- ▶ Call programs in CICS or IMS™ using J2CA APIs
- ▶ Access the network using protocols such as TCP/IP and HTTP
- ▶ Perform CPU-intensive calculations
- ▶ Integrate with other programming languages using Java Native Interface (JNI) and Language Environment® (LE)

You can also use functions in the area of security and systems management. Java on z/OS provides a number of security frameworks for security-related functionality, such as encryption, authentication, and authorization, and the JVMTI interface for systems management.

Java batch programs can be short running or long running. An example of a short-running Java program is a program performing a daily reconciliation between the contents of two data sets. An example of a long-running Java program is a continuously running server program that accepts requests from outside (through MQ or TCP/IP, for example).

1.1.1 Major enhancements in SDK 5.0

Java SDK 5.0 is the latest version of Java on z/OS and provides the following enhancements:

- ▶ General Java functionality
 - Full support for the Java 5 language specification
 - Replacement of the JVMPI by the JVM™ Tools Interface (JVMTI); refer to Chapter 9, “Java Virtual Machine Tool Interface” on page 139.
- ▶ IBM Java language extensions in the areas of:
 - ORB
 - XML and XSLT

- Security
- ▶ IBM Java runtime improvements
 - A better way of doing garbage collection (GC). There are now four possible configurable ways of doing garbage collection. Depending on the type of workload, a certain way of doing GC can be configured that leads to better performance of the Java environment; refer to “Garbage collection” on page 13.
 - The Just-In-Time (JIT) compiler has been improved in a number of areas:
 - Asynchronous compilation of methods
 - Multiple levels of optimization (five levels)
 - Profiling-driven recompilation
 - Virtual Machine improvements
 - Support for the usage of shared classes, which makes it possible to load classes just once, cache them and reuse them from multiple JVMs
 - Profiling and debugging improvements
 - RAS improvements
 - Trace engine
 - Dump engine
 - DTFJ Tooling API
- ▶ z/OS-specific enhancements

Inclusion of JZOS, which is a framework for running Java in batch jobs and accessing data more easily. Refer to Chapter 3, “Job management using JZOS” on page 25 and Chapter 7, “I/O using the JZOS toolkit API” on page 111 respectively. JZOS is a unique value-add for the z/OS version of the SDK.

Refer to a full summary of functions and features in SDK 5.0 at:

<http://www-128.ibm.com/developerworks/java/library/j-ibmjava1.html>

1.1.2 Conclusion

Using Java on z/OS for batch programs gives you the following advantages:

- ▶ Use the Java skills available in the company.

Developers of Java batch programs for z/OS do not need explicit knowledge of the z/OS environment, but only need to know how to develop a typical batch program. Modern development tools and the JZOS APIs help the developer to work with MVS data sets.

- ▶ Benefit from functionality that is not available in any other programming language.

Java provides much more than just file I/O and computing APIs.

- ▶ Benefit from the platform-independent development model using state-of-the-art IDEs on the workstation.

This model includes features such as remote deployment to z/OS and remote debugging on z/OS.

- ▶ Benefit from zAAP processors on System z9™.

Java batch programs can be run, in most cases, on the zAAP processors, if installed.

- ▶ Free Java runtime environment.

z/OS users can download or order Java for z/OS at no charge.

Java on z/OS - The basics

This chapter provides an introduction to Java on z/OS. It can be used to get started with Java on z/OS; it explains the key things you need to know. We describe the new garbage collection introduced in SDK 5.0 in more detail because we think this is one of the major changes in the SDK.

2.1 Java overview

Java has become a very popular programming language and runtime environment for building new applications on all platforms, including the zSeries and System z mainframes. Of course, in many mainframe installations new programs are created in languages such as COBOL, C/C++, or PL/I, but with the increasing usage of middle ware such as WebSphere Application Server, the Java programming model is also expanding. However, Java is not only used in relatively new middleware such as WebSphere, but also in the traditional transaction managers CICS and IMS, and DB2. And, of course, and that is the main theme of this redbook, in a stand-alone environment. Java can be used for programs submitted from a USS command line or in JCL. This form of Java is not running inside middleware or a transaction or database server.

Java's specifications are maintained by Sun™ Microsystems™, but other companies, IBM in particular, provide input for this specification. The Java language is based on a philosophy:

Develop once, run everywhere

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

- ▶ The Java programming model has been expanded with numerous APIs organized in libraries. The Java 2 Enterprise Edition (J2EE™) programming model goes even further and defines not only a syntax for writing programs, but also a way of packaging applications, and much more.
- ▶ Java programs are not compiled into persistent load modules. However, technologies have been added over the years to do compiles “on the flight” while running the applications. The first technology to support this was the Just In-Time Compiler (JIT). Later on, more technologies were included in the Java Virtual Machine (JVM) to speed up this process.
- ▶ Java, or actually the JVM, has its own memory management. The programmer does not have to worry about memory allocation and de-allocation. The JVM does this.

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

2.1.1 Java on z/OS

IBM is a major supporter and user of Java across all of the IBM computing platforms, including z/OS. The z/OS Java implementation provides the same full function Java APIs as on all other IBM platforms. In addition, the z/OS Java program products have been enhanced to allow Java access to z/OS-unique file systems. Also, Java on z/OS provides a z/OS implementation of the Java Native Interface (JNI). Read more about JNI in Chapter 6, “Java Native Interface (JNI)” on page 85.

The Java implementation on z/OS, as on other platforms, includes an Application Programming Interface (API) and a Java Virtual Machine (JVM) to run programs. The existence of a Java Virtual Machine means that applications written in Java are largely independent of the operating system being used.

Note: To stimulate to use of Java on the mainframe, IBM introduced a new specialty processor for running Java applications. It is called the zSeries Application Assist Processor, also known as zAAP. This type of processor is an optional feature in the System z9 hardware. Once installed and enabled, it allows the customer to benefit from additional resources available for Java code, and in some select cases, non-Java code closely related to the execution of Java. For zAAP processors no software license fees have to be paid for certain IBM software products. zAAP processors make it possible to expand the system’s CPU capacity at a relatively low cost, as long as the workload that is run is based on Java.

On the hardware level settings, in PR/SM™, zAAPs are treated and managed as a separate pool of logical processors. So the weight factors can be different from what you have in place for the General Purpose (GP) processors.

Note that for zAAP processors, special hardware and operating system requirements exist. For more information about the zAAP processors, see *Java Stand-alone Applications on z/OS, Volume I*, SG24-7177.

Java Software Development Kit (SDK) on z/OS

Java SDKs contain industry standard APIs. Each SDK product can be ordered and serviced independently and separately. The Java SDKs for z/OS are available electronically through the Internet, or by placing a regular software order at IBM Software Manufacturing. See the following Internet link for more information about these products and for download instructions:

<http://www-03.ibm.com/servers/eserver/zseries/software/java/allproducts.html>

At this time the following SDK products for z/OS are available:

- ▶ IBM SDK for z/OS, Java 2 Technology Edition, V1.4 (5655-I56), SDK1.4.2
- ▶ IBM 64-bit SDK for z/OS, Java 2 Technology Edition, V1.4 (5655-M30), SDK1.4.2
- ▶ IBM 31-bit SDK for z/OS, Java 2 Technology Edition, V5 (5655-N98), SDK5
- ▶ IBM 64-bit SDK for z/OS, Java 2 Technology Edition, V5 (5655-N99), SDK5

All these SDK products are available in a non-SMP/e installable flavor and an SMP/e installable flavor.

Note that, as time and technology progress, products may be withdrawn from service and new ones may be introduced.

z/OS UNIX environment

The Java Virtual Machine runs in the z/OS UNIX environment. This is largely because the JVM kernel itself is partly based on the C/C++ programming language. The combination of UNIX and C/C++ is a very natural combination.

Figure 2-1 on page 9 shows how z/OS UNIX and the Java Virtual Machine help to execute Java programs.

z/OS UNIX offers “open” interfaces for UNIX-type applications and interactive users on a z/OS system. The two z/OS UNIX software elements are:

- ▶ z/OS UNIX System Services (actually included in the z/OS MVS BCP element)
- ▶ z/OS UNIX Application Services
This is a separate z/OS base element that enhances the functionality of system services for application development.

The first element (BCP) provides a UNIX environment conforming to the XPG 4.2 standard. XPG stands for X/Open Portability Guide, which is published by the X/Open company. The guide contains all aspects of the operating system, programming languages, and protocols that compliant systems, such as z/OS, should have incorporated to call themselves UNIX systems. UNIX in z/OS is therefore fully UNIX-branded and the Assembler Callable Services of z/OS allow z/OS UNIX programs to get access to general resources. C/C++ programs use the runtime library after compilation to access these Callable Services. This functionality is provided by the z/OS UNIX kernel and the z/OS Language Environment (LE). Under the covers the JVM is doing the same thing, because for z/OS the JVM is nothing more than a regular C/C++ program.

z/OS UNIX Application Services is a separate z/OS base element, but can be considered as standard. z/OS UNIX Application Services interprets commands from users or from programs, called shell scripts, and requests MVS services in response to the commands.

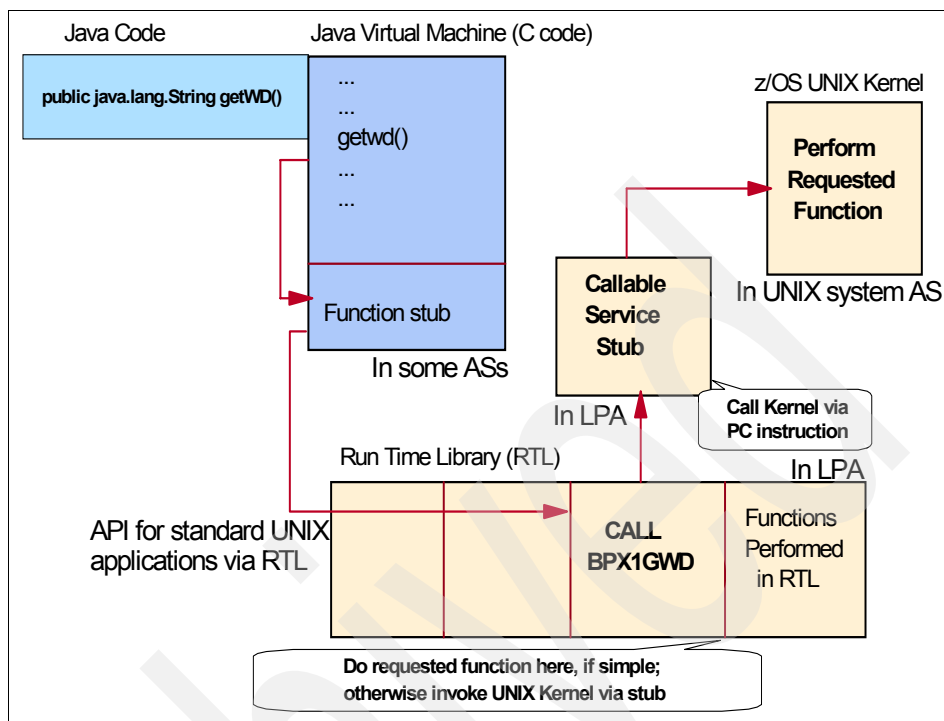


Figure 2-1 z/OS UNIX components

Verifying the SDK installation on z/OS

After the installation of the SDK, your path should contain the binary directory where Java is located. The following variables should be set or exported when using Java:

```

PATH=/usr/lpp/java/J5.0/bin
CLASSPATH=/usr/lpp/java/J5.0/lib/classes.zip
LIBPATH=/usr/lpp/java/J5.0/include
JAVA_HOME=/usr/lpp/java/J5.0/
  
```

The PATH statement is an environment variable to tell a UNIX-like system where the programs (binaries) are located, in this case for the program called java.

Now verify that java is correctly installed, as follows:

```

BRUINSM:/u/bruinsm: >java -fullversion
java full version "J2RE 1.5.0 IBM z/OS build pmz31dev-20060607 (SR2)"
BRUINSM:/u/bruinsm: >
  
```

Note: It is possible to install multiple versions of SDK products in z/OS UNIX. It is the environment variable that points to a specific version of the SDK at runtime. Each shell user can use an SDK version at any specific time, independent from other UNIX shell users.

2.1.2 Considerations when using Java on z/OS

When designing a Java application for z/OS, a key consideration is whether the application will run as a batch program or an online program. In some cases, the decision is obvious, but most applications can be designed to fit either model.

The most obvious reason for using batch is the need to process large amounts of data in a manner that does not require interaction with humans, or to run a program that does not need to provide a response back immediately. On the z/OS platform, standalone Java programs can be executed in a number of ways:

- ▶ In a UNIX shell command, optionally submitted to run in the background, by using the `&` parameter.
- ▶ In JCL running BPXBATCH to run programs in a z/OS UNIX environment.
- ▶ In JCL running BPXBATSL to run programs in a z/OS UNIX environment that access z/OS data sets.
- ▶ Using the JZOS job launcher to run programs in a z/OS UNIX environment; see Chapter 3, “Job management using JZOS” on page 25.

By “standalone” we mean in this case Java programs that only require a JVM in order to run and not Java programs that require a J2EE-compliant middleware environment such as Websphere Application Server.

Instead of executing your Java program directly from the z/OS UNIX shell prompt, it is also possible to run a Java application as a traditional batch job in z/OS. Batch jobs written in the Job Control Language (JCL) and submitted by a user run in a separate address space and can make use of the advanced workload management techniques of z/OS. Also, job schedulers can be used to submit this JCL. Although JCL is being used, the actual Java program still runs under z/OS UNIX System Services.

The utilities that can be used to execute Java programs in batch are BPXBATCH, BPXBATSL, and JZOS. BPXBATSL provides an alternate entry point in BPXBATCH and JZOS provides an entire framework for running Java in batch.

Running Java in batch is very useful for long-running and compute-intensive programs. But there is another great advantage to running Java as a batch program, and that has to do with the region size.

The region size is very much related to the JVM heap size. The heap is the runtime data area from which memory for all class instances and arrays is allocated. The heap is created at JVM startup with an initial size. Heap storage for objects is reclaimed by an automatic storage management system known as the Garbage Collector. See 2.2, “Garbage collection” on page 13.

Region size

Java programs can take up a lot of system resources, sometimes more than you may expect. Therefore, it is always necessary to examine your region size and allow for heap and stack storage requirements, plus LE and Java code, and LE internal control blocks.

The region size describes the amount of storage in which a user is allowed to run or execute programs. This value determines what kinds of programs (depending on their size) and how many programs are executable at the same time.

TSO users

In principle, TSO users inherit the region size from the logon panel for their programs. If a user enters the z/OS UNIX shell by issuing a TSO OMVS command, the parent process doesn't run in the user's address space. Under control of the WorkLoad Manager, an address space is created that gets the same region size as the TSO/E user itself.

The maximum region size you can specify on your TSO logon panel depends on the settings in your TSO profile in RACF. You can check the maximum size with the LISTUSER command:

```
TSO lu bruinsm tso
```

The LISTUSER command provides you with information about the initial logon size (SIZE) and the maximum size (MAXSIZE) you are able to specify on your logon panel.

```
TSO INFORMATION
-----
ACCTNUM= ACCNT#
PROC= IKJACCNT
SIZE= 00860000
MAXSIZE= 00000000
UNIT= SYSALLDA
USERDATA= 0000
COMMAND= ISPPDF
```

In our example you see an initial size of around 86 MB and an unlimited maximum size.

Batch users

If BPXBATCH is used, there is a REGION parameter that describes the region size:

```
//BPXBATCH EXEC PGM=BPXBATCH,REGION=0M
```

Note: It is very useful to run Java programs in a batch job, such as BPXBATCH or JZOS, because it offers good control of resource usage. Java programs can easily take more memory than you can give them when using the TSO logon panel.

Telnet rlogin and SSH users

Telnet rlogin and SSH users get their region size from the ASSIZEMAX parameter in their OMVS segment. Since OS/390 V2R8, it is possible to add individual z/OS UNIX user limits to this OMVS segment. Use the following command to display a user's OMVS segment information:

```
TSO lu bruinsm omvs
```

Following is the OMVS segment information for a specific user. This user has an ASSIZEMAX of 20 MB specified.

```
OMVS INFORMATION
-----
UID= 0000001045
HOME= /u/bruinsm
PROGRAM= /bin/sh
CPUTIMEMAX= NONE
ASSIZEMAX= 0020480000
FILEPROCMAx= NONE
PROCUSERMAX= NONE
THREADSMAX= NONE
MMAPAREAMAX= NONE
```

There is also another setting in the SYS1.PARMLIB(BPXPRMxx) member, called the MAXASSIZE parameter. MAXASSIZE is the system-wide limit and represents the maximum size for an address space created by one of the z/OS UNIX daemons.

Note: IEFUSI is a user exit where an installation can set the region size and region limit for all programs that run under this job step. Make sure this exit does not change the region size setting for the z/OS UNIX process.

Recommendations

We highly recommend to “package” your Java batch programs inside JCL, which is the case with BPXBATCH, BPXBATSL and JZOS, especially in a production environment. The reasons for this are:

- ▶ It allows you to treat this work as regular batch work, including the way it is scheduled by JES2. You will also implicitly benefit from your existing job scheduling tools.
- ▶ It allows you to set up a proper batch security environment.
- ▶ It makes it easier to get guaranteed CPU resources via WLM and memory (heap) for its JVM.
- ▶ You can direct your output to the spool for further handling or archiving. This works even better with JZOS; refer to Chapter 3, “Job management using JZOS” on page 25.
- ▶ Combining Java with other (JCL) steps in your business process can make the flow more organized.

2.2 Garbage collection

Garbage collection is part of the JVM. This task is carried out in the background to reclaim unusable space and is very important for the performance of the JVM.

The *heap size* given to a Java program can be specified using an initial and a maximum heap size parameter. An example of a user-defined heap size would look like this:

```
java -Xms20MB -Xmx35MB HelloWorld
```

Where `-Xms` specifies the initial heap size of 20 MB and `-Xmx` the maximum heap size of 35 MB. This means that our Java program will start with 20 MB of memory. If the total amount of free heap gets too low, the JVM will expand the heap size. It will continue to do so until it reaches the value specified as the maximum heap size, `-Xmx`, or if the Address Space limit has been reached based on the region size.

Note: For performance reasons it is recommended to use a fixed-size heap, which means the `-Xms` value is equal to the `-Xmx` value. This way no heap expansion or contraction occurs and this can lead to significant performance gains in some situations.

Important: We mentioned earlier that Java, and thus the JVM, can also be used from WebSphere Application Server, CICS Transaction Server, IMS Transaction Manager, DB2 on z/OS, and other tools and products. The recommended settings for the JVM may vary depending on the type of middleware or product used. We recommend that you check any middleware-specific JVM recommendations, in addition to the recommendations for the JVM itself.

When a Java program makes a request for storage and the JVM is unable to comply, an allocation failure occurs. This allocation failure triggers the Garbage Collection process. Garbage Collection is the process of automatically freeing objects that are no longer referenced by the program.

It is better to avoid very frequent Garbage Collections. Tune the Garbage Collection frequency by specifying enough heap size for your program. Giving your program too much heap size will most likely lead to other resource problems, so try to find some balance.

A good approach would be to use the *verbosegc* parameter. This parameter enables *verbosegc* logs, which you can use to determine the best heap size settings for your program. Enabling *verbosegc* can have a performance impact on an application, although it is very common for server applications to keep *verbosegc* enabled at all times.

Note: The behavior of the Garbage Collector is very much related to the heap size. A small heap may produce more frequent, but shorter GC cycles. A large heap size, on the other hand, will produce less frequent but longer GC cycles.

2.2.1 Garbage Collection in IBM SDK 5.0

The way the Java heap operates with SDK 5.0 is somewhat different from previous SDK releases. In this section we explain its functionality.

SDK 5.0 provides us with the ability to use different Garbage Collection (GC) *policies*. This means that we can influence the behavior of the GC and its corresponding algorithms, which exist under the covers of the JVM. We now have four different policies available to choose from, although the default policy will be applicable to most applications. Let us take a look at the available policies:

- | | |
|--------------------|--|
| optthruput | This is the default and should result in an optimization where throughput is our main concern. |
| optavgpause | In this case we want the GC to have more focus on the time it takes to do garbage collection. In other words, we want the GC |

to have less impact on application pause times. Large heap size, where the GC cycle takes a long time, can benefit from this policy. Or a situation where timeouts can occur with the user interface if we allow a relatively longer pause time during the GC cycle. Consider this setting when working with a 64-bit JVM and very large heap sizes.

| | |
|----------------|---|
| gencon | Applications that use many short-lived objects will see a benefit in using this policy. This policy handles short-lived objects differently and the GC cycle will do its best to comply with shorter pause times and still achieve a good throughput. This option is most likely applicable in situations that are transaction-based. |
| subpool | This policy is suitable for multiprocessor machines and multithreaded Java applications. It handles large machines somewhat better than the default policy. We are talking here about machines with 16 or more processors, so we are probably looking for tremendous scalability in this case. |

These policies can be used by specifying them as an additional parameter when starting the JVM for a specific application. The JVM parameter to get the default policy is, for example:

```
java -Xms20MB -Xmx35MB -xgcpolicy:optthruput HelloWorld
```

It is highly recommended that you start with the default policy and measure your applications' performance to evaluate this policy.

2.2.2 Detailed description of Garbage Collection

Garbage Collection is performed when an allocation failure occurs in heap lock allocation, or if a specific call to `System.gc()` occurs. The thread that has the allocation failure or the `System.gc()` call takes control and performs the Garbage Collection.

The first step is to acquire exclusive control of the Virtual Machine to prevent any further Java operations. Garbage Collection then goes through the three phases: *mark*, *sweep*, and, if required, *compaction*. The IBM Garbage Collector is a stop-the-world (STW) operation, because all application threads are stopped while the garbage is being collected.

Mark phase

In this phase, all the live objects are marked. Because unreachable objects cannot be identified individually, all the reachable objects must be identified.

Therefore, everything else must be garbage. The process of marking all reachable objects is also known as *tracing*.

The mark phase uses a pool of structures called *work packets*. Each work packet contains a *mark stack*. A mark stack contains references to live objects that have not yet been traced. Each marking thread refers to two work packets: an input packet from which references are popped and an output packet to which unmarked objects that have just been discovered are pushed. References are marked when they are pushed onto the output packet. When the input packet becomes empty, it is added to a list of empty packets and replaced by a non-empty packet. When the output packet becomes full it is added to a list of non-empty packets and replaced by a packet from the empty list.

The JVM is made up of the saved registers for each thread, the set of stacks that represents the threads, the statics that are in Java classes, and the set of local and global JNI references. All functions that are invoked in the JVM itself cause a frame on the C stack. This frame might contain references to objects as a result of either an assignment to a local variable, or a parameter that is sent from the caller. All these references are treated equally by the tracing routines.

All the mark bits for all root objects are set and references to the roots pushed to the output work packet. Tracing then proceeds by iteratively popping a reference off the marking thread's input work packet and then scanning the referenced object for references to other objects. If there are any references to unmarked objects, that is, the mark bit is off, the object is marked by setting the appropriate bit in the mark bit array and the reference is pushed to the marking thread's output work packet. This process continues until all the work packets are on the empty list, at which point all the reachable objects have been identified.

Mark stack overflow

Because the set of work packets has a finite size, it can overflow. If an overflow occurs, the Garbage Collector empties one of the work packets by popping its references one at a time, and chaining the referenced objects off their owning class by using the class pointer slot in the object header. All classes with overflow objects are also chained together. Tracing can then continue as before. If a further mark stack overflow occurs, more packets are emptied in the same way.

When a marking thread asks for a new non-empty packet and all work packets are empty, the GC checks the list of overflow classes. If the list is not empty, the GC traverses this list and repopulates a work packet with the references to the objects on the overflow lists. These packets are then processed as described above. Tracing is complete when all the work packets are empty and the overflow list is empty.

Parallel mark

The goal of *parallel mark* is to not degrade mark performance on a uniprocessor, and to increase typical mark performance on a multiprocessor system. Object marking is increased through the addition of helper threads that share the use of the pool of work packets; for example, full output packets that are returned to the pool by one thread can be picked up as new input packets by another thread. Parallel mark still requires the participation of one application thread that is used as the master coordinating agent. This thread performs very much as it always did, but the helper threads assist both in the identification of the root pointers for the collection and in the tracing of these roots. Mark bits are updated by using host machine atomic primitives that require no additional lock.

By default, a platform with n processors also has $n-1$ new helper threads, that work with the master thread to complete the marking phase of garbage collection. You can override the default number of threads by using the `-Xgcthreads` option. If you specify a value of 1, there will be no helper threads. The `-Xgcthreads` option accepts any value greater than 0, but clearly you gain nothing by setting it to more than $N-1$.

Concurrent mark

Concurrent mark gives reduced and consistent garbage collection pause times when heap sizes increase. It starts a concurrent marking phase before the heap is full. In the concurrent phase, the Garbage Collector scans the roots, i.e. stacks, JNI references, class statics, etc. The stacks are scanned by asking each thread to scan its own stack. These roots are then used to trace live objects concurrently. Tracing is done by a low-priority background thread and by each application thread when it does a heap lock allocation.

While the Garbage Collector is marking live objects concurrently with application threads running, it has to record any changes to objects that are already traced. It uses a write barrier that is activated every time a reference in an object is updated.

The write barrier flags when an object reference update has occurred, to force a re-scan of part of the heap. The heap is divided into 512-byte sections and each section is allocated a byte in the card table. Whenever a reference to an object is updated, the card that corresponds to the start address of the object that has been updated with the new object reference is marked with 0x01. A byte is used instead of a bit to eliminate contention; it allows marking of the cards using non-atomic operations. An STW collection is started when one of the following occurs:

- ▶ An allocation failure
- ▶ A `System.gc`
- ▶ Concurrent mark completes all the marking that it can do

Garbage Collector tries to start the concurrent mark phase so that it completes at the same time as the heap is exhausted. The Garbage Collector does this by constant tuning of the parameters that govern the concurrent mark time. In the STW phase, the Garbage Collector rescans all roots and uses the marked cards to see what else must be retraced, and then sweeps as normal. It is guaranteed that all objects that were unreachable at the start of the concurrent phase are collected.

It is not guaranteed that objects that become unreachable during the concurrent phase are collected. Objects which become unreachable during the concurrent phase are referred to as “floating garbage”. Reduced and consistent pause times are the benefits of concurrent mark, but they come at a cost. Application threads must do some tracing when they are requesting a heap lock allocation. The overhead varies depending on how much idle CPU time is available for the background thread. Also, the write barrier has some overhead.

The -Xgcpolicy command-line parameter is used to enable and disable concurrent mark:

```
-Xgcpolicy:<optthruput | optavgpause | gencon | subpool>
```

The -Xgcpolicy options have these effects:

| | |
|--------------------|--|
| optthruput | Disables concurrent mark. If you do not have pause time problems (as seen by erratic application response times), you get the best throughput with this option. Optthruput is the default setting. |
| optavgpause | Enables concurrent mark with its default values. If you are having problems with erratic application response times that are caused by normal garbage collections, you can reduce those problems at the cost of some throughput, by using the optavgpause option. |
| gencon | Requests the combined use of concurrent and generational GC to help minimize the time that is spent in any garbage collection pause. |
| subpool | Disables concurrent mark. It uses an improved object allocation algorithm to achieve better performance when allocating objects on the heap. This option might improve performance on SMP systems with 16 or more processors. The subpool option is available only on AIX®, Linux PPC and zSeries, z/OS, and i5/OS®. |

Sweep phase

On completion of the mark phase the mark bit vector identifies the location of all the live objects in the heap. The sweep phase uses this to identify those chunks

of heap storage that can be reclaimed for future allocations; these chunks are added to the pool of free space. To avoid filling this free space pool with lots of small chunks of storage, only chunks of at least a certain size are reclaimed and added to the free pool. The minimum size for a free chunk is currently defined as 512 bytes (768 bytes on 64-bit platforms).

A free chunk is identified by examining the mark bit vector looking for sequences of zeros, which identify possible free space. GC ignores any sequences of zeroes that correspond to a length less than the minimum free size. When a sequence of sufficient length is found, the Garbage Collector checks the length of the object at the start of the sequence to determine the actual amount of free space that can be reclaimed. If this amount is greater than or equal to the minimum size for a free chunk, it is reclaimed and added to the free space pool.

The small areas of storage that are not on the freelist are known as “dark matter”, and they are recovered when the objects that are next to them become free, or when the heap is compacted. It is not necessary to free the individual objects in the free chunk, because it is known that the whole chunk is free storage. When a chunk is freed, the Garbage Collector has no knowledge of the objects that were in it.

Parallel bitwise sweep

Parallel bitwise sweep improves the sweep time by using available processors. In parallel bitwise sweep, the Garbage Collector uses the same helper threads that are used in parallel mark, so the default number of helper threads is also the same and can be changed with the `-Xgcthreads` option. The heap is divided into sections of 256 KB and each thread (helper or master) takes a section at a time and scans it, performing a modified bitwise sweep. The results of this scan are stored for each section. When all sections have been scanned, the freelist is built.

Concurrent sweep

Like concurrent mark, concurrent sweep gives reduced garbage collection pause times when heap sizes increase. Concurrent sweep starts immediately after a stop-the-world (STW) collection, and must at least complete a certain subset of its work before concurrent mark is allowed to kick off, because the mark map used for concurrent mark is also used for sweeping, as described previously. The concurrent sweep process is split into two types of operations:

- ▶ Sweep analysis, which processes sections of data in the mark map and determines ranges of free or potentially free memory to be added to the free list
- ▶ Connection, which takes analyzed sections of the heap and connects them into the freelist sections are calculated in the same way as for parallel bitwise sweep.

Heap

An STW collection initially performs a minimal sweep operation that searches for and finds a free entry large enough to satisfy the current allocation failure. The remaining unprocessed portion of the heap and mark map are left to concurrent sweep to be both analyzed and connected. This work is accomplished by Java threads through the allocation process. For a successful allocation, an amount of heap relative to the size of the allocation is analyzed, and is performed outside the allocation lock. Within an allocation, if the current free list cannot satisfy the request, sections of analyzed heap are found and connected into the free list. If sections exist but are not analyzed, the allocating thread must also analyze them before connecting.

Because the sweep is incomplete at the end of the STW collection, the amount of free memory reported (through verbose GC or the API) is an estimate based on past heap occupancy and the ratio of unprocessed heap size against total heap size.

In addition, the mechanics of compaction require that a sweep be completed before a compaction can occur. Consequently, an STW collection that compacts will not have concurrent sweep active during the next round of execution.

To enable concurrent sweep, use the `-Xgcpolicy:` parameter `optavgpause`. It is active in conjunction with concurrent mark. The modes `optthruput`, `gencon`, and `subpool` do not support concurrent sweep.

Compaction phase

When the garbage has been removed from the heap, the Garbage Collector can consider compacting the resulting set of objects to remove the spaces that are between them. The process of compaction is complicated because, if any object is moved, the Garbage Collector must change all the references that exist to it. The default is not to compact. The following analogy might help you understand the compaction process. Think of the heap as a warehouse that is partly full of pieces of furniture of different sizes. The free space is the gaps between the furniture. The free list contains only gaps that are above a particular size. Compaction pushes everything in one direction and closes all the gaps. It starts with the object that is closest to the wall, and puts that object against the wall. Then it takes the second object in line and puts that against the first. Then it takes the third and puts it against the second, and so on. At the end, all the furniture is at one end of the warehouse and all the free space is at the other. To continue the analogy, in the Java SDK 1.4.2, some of the boxes were nailed to the floor. In Java 5.0 we can perform a full compaction. The immovable reference restrictions are removed.

To keep compaction times to a minimum, the helper threads are used again. Compaction occurs if any one of the following is true and `-Xnocompactgc` has not been specified:

- ▶ `-Xcompactgc` has been specified.
- ▶ Following the sweep phase, not enough free space is available to satisfy the allocation request.
- ▶ A `System.gc()` has been requested and the last allocation failure garbage collection did not compact or `-Xcompactexplicitgc` has been specified.
- ▶ At least half the previously available memory has been consumed by TLH allocations (ensuring an accurate sample) and the average TLH size falls below 1024 bytes.
- ▶ Less than 5% of the active heap is free.
- ▶ Less than 128 KB of the active heap is free.

Reference objects

When a reference object is created, it is added to a list of reference objects of the same type. The referent is the object to which the reference object points. Instances of *SoftReference*, *WeakReference*, and *PhantomReference* are created by the user and cannot be changed; they cannot be made to refer to objects other than the object that they referenced on creation. Objects whose classes define a `finalize` method result in a pointer to that object being placed on a list of objects that require finalization.

During garbage collection, immediately following the mark phase, these lists are processed in a specific order:

1. Soft
2. Weak
3. Final
4. Phantom

Soft, weak, and phantom reference processing

For each element on a list, GC determines if the reference object is eligible for processing and then if it is eligible for collection. An element is eligible for processing if it is marked and has a non-null referent field. If this is not the case, the reference object is removed from the reference list, resulting in it being freed during the sweep phase.

If an element is determined to be eligible for processing, GC must determine if it is eligible for collection. The first criterion here is simple: Is the referent marked?

If it is marked, the reference object is not eligible for collection and GC moves onto the next element of the list.

If the referent is not marked, GC has a candidate for collection. At this point the process differs for each reference type. Soft references are collected if their referent has not been marked for the previous 32 garbage collection cycles. You adjust the frequency of collection with the `-Xsoftrefthreshold` option. If there is a shortage of available storage, all soft references are cleared. All soft references are guaranteed to have been cleared before the `Out_Of_Memory` error is thrown.

Weak and phantom references are always collected if their referent is not marked. When a phantom reference is processed, its referent is marked so it will persist until the following garbage collection cycle or until the phantom reference is processed if it is associated with a reference queue. When it is determined that a reference is eligible for collection, it is either queued to its associated reference queue or removed from the reference list.

Final reference processing

The processing of objects that require finalization is more straightforward. The list of objects is processed and any element that is not marked is processed by marking and tracing the object and then creating an entry on the finalizable object list for the object. Then GC removes the element from the unfinalized object list. The final method for the object is then run at an undetermined point in the future by the reference handler thread.

JNI weak reference

JNI weak references provide the same capability as that of `WeakReference` objects, but the processing is very different. A JNI routine can create a JNI weak reference to an object and later delete that reference. The Garbage Collector clears any weak reference where the referent is unmarked, but no equivalent of the queuing mechanism exists. Note that failure to delete a JNI weak reference causes a memory leak in the table and performance problems. This is also true for JNI global references. The processing of JNI weak references is handled last in the reference handling process. The result is that a JNI weak reference can exist for an object that has already been finalized and had a phantom reference queued and processed.

Heap expansion

Heap expansion occurs after Garbage Collection while exclusive access of the virtual machine is still held. The active part of the heap is expanded up to the maximum if one of the following is true:

- The Garbage Collector did not free enough storage to satisfy the allocation request.

- ▶ Free space is less than the minimum free space, which you can set with the `-Xminf` parameter. The default is 30%.
- ▶ More than the maximum time threshold is being spent in garbage collection, set using the `-Xmaxt` parameter. The default is 13%.

The amount to expand the heap is calculated as follows:

- ▶ If the heap is being expanded because less than `-Xminf` (default 30%) free space is available, the Garbage Collector calculates how much the heap needs to expand to get `-Xminf` free space.
- ▶ If this is greater than the maximum expansion amount, which you can set with the `-Xmaxe` parameter (default of 0, which means no maximum expansion), the calculation is reduced to `-Xmaxe`.
- ▶ If this is less than the minimum expansion amount, which you can set with the `-Mine` parameter (default of 1 MB), it is increased to `-Mine`.
- ▶ If the heap is expanding and the JVM is spending more than the maximum time threshold, the Garbage Collector calculates how much expansion is needed to expand the heap by 17% free space. This is adjusted as above, depending on `-Xmaxe` and `-Mine`.
- ▶ Finally, the Garbage Collector ensures that the heap is expanded by at least the allocation request if garbage collection did not free enough storage.

All calculated expansion amounts are rounded up to a 512-byte boundary on 32-bit JVM or a 1024-byte boundary on 64-bit JVMs.

Heap shrinkage

Heap shrinkage occurs after garbage collection while exclusive access of the virtual machine is still held. Shrinkage does not occur if any of the following are true:

- ▶ The Garbage Collector did not free enough space to satisfy the allocation request.
- ▶ The maximum free space, which can be set by the `-Xmaxf` parameter (default is 60%), is set to 100%.
- ▶ The heap has been expanded in the last three garbage collections.
- ▶ This is a `System.gc()` and the amount of free space at the beginning of the garbage collection was less than `-Xminf` (default is 30%) of the live part of the heap.
- ▶ If none of the above is true and more than `-Xmaxf` free space exists, the Garbage Collector must calculate how much to shrink the heap to get it to `-Xmaxf` free space, without going below the initial (`-Xmas`) value. This figure is

rounded down to a 512-byte boundary on 32-bit JVMs or a 1024-byte boundary on 64-bit JVMs.

A compaction occurs before the shrink if all the following are true:

- ▶ A compaction was not done in this garbage collection cycle.
- ▶ No free chunk is at the end of the heap, or the size of the free chunk that is at the end of the heap is less than 10% of the required shrinkage amount.
- ▶ The Garbage Collector did not shrink and compact in the last garbage collection cycle.

Note that, on initialization, the JVM allocates the whole heap in a single contiguous area of virtual storage. The amount that is allocated is determined by the setting of the -Xmx parameter. No virtual space from the heap is ever freed back to the native operating system. When the heap shrinks, it shrinks inside the original virtual space.

You never see the amount of virtual memory that is used by the JVM decrease. You might see physical memory free size increase after a heap shrinkage.

Additional memory is committed as the heap grows.

Job management using JZOS

In this chapter, we discuss the positioning of JZOS compared to other Java batch job launcher techniques described in *Java Stand-alone Applications on z/OS, Volume I*, SG24-7177 (BPXBATCH and BPXBATSL), and present examples to illustrate the strength of the additional features available in the JZOS job launcher. The fundamentals of job management on z/OS are covered in Chapter 2 of *Java Stand-alone Applications on z/OS, Volume I*, SG24-7177. We highly recommend that you read this chapter before proceeding with this chapter if you are new to the z/OS environment.

Refer to

<http://dovetail.com>

for more details on examples using JZOS.

3.1 Positioning of JZOS

BPXBATCH and *BPXBATSL* have been the tools of choice for running executable file and shell scripts residing in the Hierarchical File System (HFS) under UNIX System Services from a TSO session and z/OS Job Control Language (JCL). We described how to use these tools to run Java applications as batch jobs in Chapter 2, “Job management” in *Java Stand-alone Applications on z/OS, Volume I*, SG24-7177.

These tools are useful and flexible in their own right. However, they have some limitations from the view of traditional batch job management in z/OS systems. A batch job typically runs for a long time, processing a large volume of data in a controlled environment. Although such jobs are not designed to be interactive, operators need to be able to monitor the progress of jobs and take any necessary action when a job requires special attention.

That is, jobs need to be reliable and recoverable when they encounter problems, preferably without stopping the jobs (console communication). Moreover, a job typically consists of multiple steps and their executions depend on each other. In some cases, the job executes a different set of programs depending on the results of previous steps in the job (condition passing between job steps). Resource requirements for a job are specified in a JCL script (DD statements). It makes the job more flexible. For example, it is possible to allocate temporary data sets in the context of a job and they are then shared across the job steps.

Having discussed the typical characteristics and requirements for batch jobs, some shortfalls between them and *BPXBATCH* and *BPXBATSL* (plus runtime APIs in the SDK) are made obvious. The JZOS job launcher and runtime APIs are the latest inclusion in the IBM Java SDK distribution for z/OS, which addresses these shortfalls¹.

Firstly, JZOS includes Java classes that make the console communication from Java applications easy. Instead of implementing the console communication via Java Native Interface (JNI) calls, JZOS provides a framework to register a *listener* for interaction with operators (WTO). It also provides a method to write messages to MVS system logs directly from Java applications. Using this framework, Java programs are able to write messages to the z/OS system log when they require special attention from the operator. This can be used as a means to interface with the system automation tools to monitor the status of the running Java batch programs. It is also possible to receive commands from the operator while Java batch jobs are running (accepting a **modify** command). Consequently, it is possible to program Java programs to change behavior

¹ IBM acquired JZOS batch technology from Dovetailed Technologies, LLC in 2006. At the time of writing, JZOS was available from alphaWorks® at no charge. Currently, JZOS is part of the standard distribution of IBM Java SDK 5.0 for z/OS and the IBM Java SDK 1.4.2 SR6 for z/OS.

depending on the commands received from the operator without restarting the job. 3.4, “Console communication with batch jobs” on page 42 describes how to make Java applications ready for console communication.

Secondly, JZOS enables Java programs to be integrated seamlessly with other job steps within a job. Execution of job steps is typically controlled by *return codes* from the previous job steps. JZOS reliably delivers the exit status of Java programs to the following steps in a job, which allows for inclusion of Java program steps in other MVS utilities and programs.

Another big advantage of JZOS is its full support for DD statements. Because Java programs launched via JZOS run in the same address space as any other steps, Java programs are able to access DD statements specified in a job. For example, it is common to allocate a temporary data set to store output from a step and be used by the following steps in a job. Using JZOS, Java programs are able to read and write from such data sets, which makes Java programs more appealing when considering replacing legacy applications. 3.3, “Art of job management using JZOS” on page 30 discusses how to use JZOS to include Java programs as part of batch jobs in more detail.

Thirdly, the JZOS batch launcher directs stdout and stderr input streams to the standard MVS data sets and the JESS SYSOUT data set. Also, Java programs are able to read from stdin from the standard MVS data sets. With BPXBATCH, it is only possible to write/read to files in USS. Consequently, operators are able to monitor the output from Java programs via System Display and Search Facility (SDSF), just like monitoring any other job steps in MVS environments.

Table 3-1 summarizes the differences between BPXBATCH, BPXBATSL, and JZOS. Although JZOS offers many additional features that are desirable to integrate Java programs in batch jobs, it is not intended to replace BPXBATCH and BPXBATSL. For instance, a job that requires to run shell scripts or programs residing in USS will continue to use BPXBATH and BPXBATSL.

Table 3-1 Comparison between BPXBATCH, BPXBATSL, and JZOS

| | BPXBATCH | BPXBATSL | JZOS |
|---|----------|----------|------|
| Run in the same address space | No | Yes | Yes |
| DD statements supported | No | Yes | Yes |
| stdin, stdout, and stderr to MVS data set | No | No | Yes |

| | BPXBATCH | BPXBATSL | JZOS |
|---|-----------------------------|--------------------|------|
| Console communication | No | Yes, via JNI calls | Yes |
| Return code (System.exit) | Yes, but always 1 otherwise | RC multiply by 256 | Yes |
| Running programs and shell scripts in USS | Yes | Yes | No |

3.2 JZOS launcher settings and return codes

Table 3-2 summarizes DD names that are used by the JZOS batch launcher.

Table 3-2 DD names used by JZOS batch launcher

| DD name | Description | Required |
|----------|---|----------|
| SYSOUT | Messages from the batch launcher and any system messages that are written to the UNIX stderr file descriptor. | YES |
| SYSPRINT | An system messages which are written to the UNIX stdout file descriptor. This is normally used. | NO |
| STDOUT | The output from Java <i>System.out</i> . This data is translated to the JZOS_OUTPUT_ENCODING codepage. | YES |
| STDERR | The output from Java <i>System.err</i> . This data is translated to the JZOS_OUTPUT_ENCODING codepage. | YES |
| STDENV | A UNIX shell script used to configure environmental variables. | YES |
| STDIN | The input to Java <i>System.in</i> . This data is translated from the JZOS_OUTPUT_ENCODING codepage to the default Java file.encoding codepage. | NO |
| MAINARGS | Can be used to supply arguments to the main Java class. | NO |

Table 3-3 on page 29 summarizes additional environmental variables that control the behavior of JZOS and Java applications launched via the JZOS job launcher.

Table 3-3 Description of JZOS environmental variables

| Environmental Variables | Description |
|--|---|
| JZOS_ENABLE_MVS_COMMANDS = { <u>true</u> false} | This environmental variable determines whether or not JZOS will allow processing of the MVS operator commands START(S), MODIFY(F) and STOP (P). If set to 'false', the JZOS batch launcher will not respond to MVS operator commands. The default is true. |
| JZOS_OUTPUT_ENCODING = {codepage} | This environmental variable specifies the codepage used by JZOS for its output to stdout and stderr. If not specified, the default codepage for the current locale is used (IBM-1047 EBCDIC codepage). |
| JZOS_ENABLE_OUTPUT_TRANSCODING = { <u>true</u> false} | If set to false, raw bytes written to System.out and System.err are not transcoded to the JZOS_OUTPUT_ENCODING codepage. |
| JZOS_GENERATE_SYSTEM_EXIT = {true <u>false</u> } | If set to true, JZOS will generate a System.exit() call upon completion of main(). This will cause JZOS to complete, even if there are active non-daemon threads. The default is false, which means JZOS will wait for non-daemon threads to complete before exiting. |
| JZOS_MAIN_ARGS = {classname and arguments} JZOS_MAIN_ARGS_DD = {ddname main args} | Allows for additional arguments for the main method that JZOS invokes. |

Table 3-4 on page 30 shows all return codes currently used by JZOS. To prevent a Java exit code from matching a JZOS return code, it is recommended *not* to use the range between 100 to 102.

When the batch launcher completes normally, it issues a return code.

- If the launcher itself fails, SYSOUT will contain the message:


```
JVMJZBL1021E JZOS batch launcher failed, return code=nnn
```

 where nnn is one of the codes described in Table 3-4 on page 30.
- If the launcher completes without an internal error, the return code set by Java will be returned and SYSOUT will contain the message:


```
JVMJZBL1021N JZOS batch launcher completed, return code=0
```

Table 3-4 JZOS return codes

| RC | Name | Notes |
|-----|-------------------|---|
| 0 | RC_OK | The Java main() method invoked by the launcher returned normally, or a System.exit() or System.exit(0) message was used to shut down the JVM. |
| 100 | RC_MAIN_EXCEPTION | The Java main class not found or main method threw an exception. |
| 101 | RC_CONFIG_ERR | A configuration or setup error occurred. Check SYSOUT messages for more diagnostic information. |
| 102 | RC_SYSTEM_ERR | A system or internal error occurred. Check SYSOUT messages for more diagnostic information. |

Table 3-5 summarizes different logging levels for the JZOS batch launcher. The logging messages are written to SYSOUT. The logging information can be useful to determine the cause of various errors and problems.

Table 3-5 Batch launcher logging levels

| Level | Description |
|-------|---|
| +E | Only error level messages are issued. |
| +W | Adds warning level messages. |
| +N | Adds notice level messages (the default). |
| +I | Adds informational messages. This level includes a dump of the environment variables (including CLASSPATH) prior to creating the Java VM. |
| +D | Adds debugging level messages. This level will print input to and output from the //STDENV configuration script process. |
| +T | Adds trace level messages. This level should be used when reporting a launcher problem to IBM. |

3.3 Art of job management using JZOS

In this section, we present a simple example of how to integrate Java applications as part of a traditional MVS batch job, and illustrate how Java

applications can be integrated seamlessly with other job steps and MVS utilities using the JZOS job launcher.

Example 3-1 shows a JCL script that integrates traditional batch job steps with a Java application. In this job, random data is produced by a REXX exec (called GENRCDS). The output of the exec is stored in a temporary data set which is allocated as part of the job and exists only during the execution of the job (called REXXOUT). The data is sorted using the MVS utility SORT, and the output is stored in a temporary data set again (SORTDATA).

Example 3-1 Batch job including MVS utilities, REXX exec, and Java application

```
//JZOEXPL JOB (ITSO),'KENICHI',REGION=300M,
//          CLASS=A,MSGCLASS=X,NOTIFY=&SYSUID
//PROCLIB JCLLIB ORDER=KENICHI.JZOS.SAMPJCL
//*****
/* EXECUTE REXX EXEC TO WRITE DATA TO A TEMPORARY
/* DATA SET CALLED REXXOUT.
//*****
//GENREC   EXEC PGM=IRXJCL,PARM='GENRCDS'
//SYSTSIN  DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//OUTFILE  DD DSN=&&REXXOUT,DISP=(NEW,PASS),
//          SPACE=(TRK,(2,1))
//SYSEXEC  DD DSN=KENICHI.EXECS,DISP=SHR
//*****
/* NOW EXECUTE THE SORT COMMAND AGAINST THE
/* TEMPORARY DATA SET (REXXOUT) AND WRITE RESULTS
/* TO ANOTHER TEMPORARY DATA SET (SORTDATA).
//*****
//SORTSTEP EXEC PGM=SORT
//SYSIN DD *
          SORT FIELDS=(1,80,CH,A)
/*
//SYSOUT DD SYSOUT=*
//SORTIN  DD DSN=&&REXXOUT,DISP=(OLD,DELETE)
//SORTOUT DD DSN=&&SORTDATA,DISP=(NEW,PASS),
//          SPACE=(TRK,(2,1))
/*
//*****
/* NOW CALL THE JAVA PROGRAM USING JZOS. JAVA
/* PROGRAM READS FROM THE TEMPORARY DATA SET
/* (SORTDATA), AND WRITE TO A PERMANENT FILE
/* (KENICHI.DATA).
//*****
//JAVA EXEC PROC=JVMPC50,
```

```

// JAVACLS='ProcessRecords'
//INDATA DD DSN=&&SORTDATA,DISP=(OLD,DELETE)
//OUTDATA DD DSN=KENICHI.DATA,DISP=SHR
//STDOUT DD SYSOUT=*
//STDERR DD SYSOUT=*
//STDENV DD DSN=KENICHI.JAVAENV(JPRPTY50),DISP=SHR
//*****
/* DELETE THE BACKUP COPY FROM THE PREVIOUS RUN.
//*****
//DELETE EXEC PGM=IDCAMS,REGION=1M,COND=(7,LT)
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
DELETE KENICHI.DATA.BACKUP
SET MAXCC=0
/*
//*****
/* NOW MAKE A BACKUP COPY OF THE RESULTS IF PREVIOUS
/* STEPS WERE SUCCESSFUL.
//*****
//BACKUP EXEC PGM=IEBGENER,COND=(7,LT)
//SYSPRINT DD SYSOUT=*
//SYSIN DD DUMMY
//SYSUT1 DD DSN=KENICHI.DATA,DISP=SHR
//SYSUT2 DD DSN=KENICHI.DATA.BACKUP,DISP=(NEW,CATLG),
// SPACE=(TRK,(2,1))

```

The next step (JAVA) invokes the Java application that writes the output of previous steps (that is, random records sorted alphabetically) to a permanent data set. The output from the previous sort step is made available to this step via the DD statement INDATA. The DD statement refers to a temporary data set that was allocated as part of the previous job step. && denotes a temporary data set from the previous job steps, and the DELETE parameter for the displacement of the data set indicates that the temporary data set can be removed after this step. Similarly, the name of the permanent data set is specified in the DD statement OUTDATA.

The values referenced via DD statements are made available to Java programs invoked by the JZOS job launcher in the same way DD statements are used in other traditional mainframe languages (for example, the REXX data set I/O library only accepts references to data sets via DD statements or allocated via TSO command).

In this particular example, the environmental variable for the Java program is configured in a member of data set KENICHI.JAVAENV(JPRPTY50). The runtime configuration can also be specified in a sequential file, a file in USS, or a

JCL script. Example 3-2 shows a sample script to configure the Java runtime environment. Note that the IBM_JAVA_OPTIONS (IBM Java SDK option) environmental variable is set in this script. Many options such as performance tuning and remote debugging can be specified here.

Example 3-2 Example of shell script to configure Java runtime environment

```
# This is a shell script which configures
# any environment variables for the Java JVM 5.0.
# Variables must be exported to be seen by the launcher.

. /etc/profile
export JZOS_HOME=/u/kenichi/jzos
export JAVA_HOME=/usr/lpp/java/J5.0

export PATH=/bin:${JAVA_HOME}/bin:

LIBPATH=/lib:/usr/lib:${JAVA_HOME}/bin
LIBPATH="$LIBPATH":${JAVA_HOME}/bin/classic
LIBPATH="$LIBPATH":${JZOS_HOME}
export LIBPATH="$LIBPATH":

# Customize your CLASSPATH here
CLASSPATH=/u/kenichi/examples

# Add JZOS required jars to end of CLASSPATH
for i in "${JZOS_HOME}"/*.jar; do
    CLASSPATH="$CLASSPATH":$i
done
export CLASSPATH="$CLASSPATH":

# Set JZOS specific options
# Use this variable to specify encoding for DD STDOUT and STDERR
#export JZOS_OUTPUT_ENCODING=Cp1047
# Use this variable to prevent JZOS from handling MVS operator commands
#export JZOS_ENABLE_MVS_COMMANDS=false
# Use this variable to supply additional arguments to main
#export JZOS_MAIN_ARGS=""

# Configure JVM options
IJO="-Xms16m -Xmx128m"
# Shared class option
IJO="$IJO -Xscmx25m -Xshareclasses:name=MyCache"
# Uncomment the following to aid in debugging "Class Not Found"
# problems
```

```

#IJO="$IJO -verbose:class"
IJO="$IJO -Djzos.home=${JZOS_HOME}"
# Uncomment the following if you want to run without JIT
#IJO="$IJO -Djava.compiler=NONE"
# Uncomment the following if you want to run with Ascii file encoding..
#IJO="$IJO -Dfile.encoding=ISO8859-1"
export IBM_JAVA_OPTIONS="$IJO "

export JAVA_DUMP_HEAP=false
export JAVA_PROPAGATE=NO
export IBM_JAVA_ZOS_TDUMP=NO

```

Example 3-3 illustrates a Java program that performs I/O operations using the values set via DD statements in a JCL script. For further discussion of how to access MVS data set from Java programs, refer to Chapter 3, “Access to MVS data sets” on page 31 in *Java Stand-alone Applications on z/OS, Volume I*, SG24-7177 or Chapter 7, “I/O using the JZOS toolkit API” on page 111.

If the Java step is successful (RC less than 8), it creates a backup copy of the data just written by the Java program using MVS utilities IDCAMS and IEBGENER. IDCAMS is used to delete the existing data set. IEBGENER is used to copy the data set written by the Java program to a backup data set.

Example 3-3 Example of Java program perform I/O using DD statements

```

import com.ibm.jzos.*;

public class ProcessRecords {
    public static void main(String args[]) {
        ZFile inFile = null, outFile = null;
        byte[] buffer = null;
        int nRead;

        try {
            // References to input/output data sets via DD statements.
            inFile= new ZFile("//DD:INDATA", "rb,type=record,noseek");
            outFile = new ZFile("//DD:OUTDATA", "wb,type=record,noseek");

            // Allocate buffer depending on the record length of
            // the specified input data set.
            buffer = new byte[inFile.getLrecl()];
            String enc = ZUtil.getDefaultPlatformEncoding();

            // Read records as long as more records available.
            while ((nRead = inFile.read(buffer)) >= 0) {

```



```

        // Construct string based on the record read and the
        // default platform encoding.
        String line = new String(buffer, 0, nRead, enc);
        System.out.println(line);

        // Write to the specified output data set.
        outFile.write(buffer);
        outFile.flush();
    }
} catch (Exception e) {
    // Prints the stack trace of the exception to standard
    // error output.
    e.printStackTrace();
    // Important to quit with the right RC.
    System.exit(8);
} finally {
    // Close the input file.
    if (inFile != null) {
        try {
            inFile.close();
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(8);
        }
    }

    // Close the output file.
    if (outFile != null) {
        try {
            outFile.close();
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(8);
        }
    }
}
System.out.println("Done!");
// Important to quit with the right RC.
System.exit(0);
}
}

```

This simple example highlights several important advantages offered by the JZOS job launcher:

- ▶ A Java job step can be seamlessly integrated with other job steps.
 - Usage of temporary data sets for I/O
 - Conditional execution of job steps including Java steps
 - References to data sets using DD statements
- ▶ Leverage the existing MVS utilities with Java programs.
Usage of SORT, IDCAMS, IEBGENER and much more.
- ▶ Standard I/O streams are written to standard data sets. Operators can observe the output from the job while the job is running (more discussion in 3.4, “Console communication with batch jobs” on page 42).

3.3.1 Best practice considerations

In this section, we discuss some of the best practice considerations when working with Java stand-alone applications on z/OS.

Managing return codes from Java job steps

Return codes returned from each job step within a job are important for controlling the execution of a job, which typically consists of several steps. For example, the *delete* and *backup* steps in Example 3-1 on page 31 are only executed when return codes from the previous steps are less than 8¹. When a Java application is launched via the JZOS launcher, the value specified in the `System.exit()` call is passed back as return code for the job step. Hence it is important to specify appropriate values when the Java application is terminating.

We studied the return code from a Java job step using the JZOS launcher when the Java application terminates differently. The results are summarized below:

- ▶ Terminating without explicit `System.exit()` call: RC = 0.
- ▶ Terminating when a specified exception is thrown from main method (that is, main method declared to throw *java.lang.Exception*): RC = 124.
- ▶ Terminating when an unexpected exception is thrown (that is, an unchecked exception is thrown): RC = 124.
- ▶ Terminating with explicit `System.exit(8)` call: RC = 8.
- ▶ Terminating with explicit `System.exit(0)` call: RC = 0.

From the experiments, it is safe to mix Java programs that do not have explicit `System.exit()` calls with other traditional job steps in a JCL script to control the

¹ In an MVS environment, return code 0 indicates successful execution, 4 indicates successful execution with warnings, and 8 or greater indicates execution failure.

execution (that is, at least RC 124 is returned when an unexpected exception is thrown in the program). However, it is more desirable to agree on RCs in advance so that MVS system programmers can take appropriate actions when erroneous RCs are returned from Java steps.

The JZOS product version occupies certain return code ranges. It is recommended not to use any return code mentioned in Table 3-4 on page 30 to distinguish between failure of JZOS and failure of the Java application.

Using shared classes for performance improvement

Java class sharing in the IBM Version 5.0 SDK offers a completely transparent and dynamic means of sharing all loaded classes, both application classes and system classes. It can improve the overall performance of Java applications when the same Java application is started and stopped frequently, or multiple instances of the same class are deployed. It also reduces the size of the virtual memory footprint, especially when multiple JVM instances coexist. It is an optional feature that must be turned on explicitly. Key points about the Java class sharing feature are as follows:

- ▶ Classes are stored in a named “class cache”, which is an area of shared memory of fixed size, allocated by the first JVM that needs to use it.
- ▶ Any JVM can read from or update the cache, although a JVM can connect to only one cache at a time.
- ▶ The cache persists beyond the lifetime of any JVM connected to it, until it is explicitly destroyed or until the operating system is shut down.
- ▶ When a JVM loads a class, it looks first for the class in the cache to which it is connected and, if it finds the class it needs, it loads the class from the cache. Otherwise, it loads the class from disk and adds it to the cache.
- ▶ When a cache becomes full, classes in the cache can still be shared, but no new classes can be added.
- ▶ Because the class cache persists beyond the lifetime of any JVM connected to it, some classes in the cache might become out of date if changes are made to classes on the file system. This situation is managed transparently. The updated version of the class is detected by the next JVM that loads it and the class cache is updated.
- ▶ Sharing of bytecode that is modified at runtime is supported, but must be used with care.
- ▶ Access to the class cache is protected by Java permissions if a security manager is installed.
- ▶ Only class data that does not change can be shared. Resources, objects, JIT'd code, and similar items cannot be stored in the cache.

- Options to enable shared classes are as follows:

- `-Xscmx<size>[klmlg]`

Specifies cache size. This option applies only if a cache is being created and no cache of the same name exists. The default caching size is platform-dependent. You can find out the size value being used by adding “-verbose:sizes” as a command line argument. The minimum cache size is 4 KB. The maximum cache size is platform-dependent. The size of the cache you can specify is limited by the amount of physical memory and paging space available to the system. Because the virtual address space of a process is shared between the shared classes cache and the Java heap, increasing the maximum size of the Java heap will reduce the size of the shared classes cache you can create.

- `-Xshareclasses:<suboptions>`

Enables class sharing. Can take a number of suboptions, some of which are cache utilities. Cache utilities perform the required operation on the specified cache, without starting the JVM. You can combine multiple suboptions, separated by commas. A subset of suboptions is as follows:

- `name=<name>`
Connects to a cache of a given name, creates the cache if it does not already exist.
- `verbose`
Gives detailed output on the cache I/O activity, listing information on classes being stored and found.
- `destroy` and `destroyAll`
Tries to destroy the specified cache or all caches on the system. A cache can be destroyed only if all JVMs using it have shut down, and the user has sufficient permissions.

The following example sets up a cache for shared classes with the name of MyCache and the maximum size of 25 MB. Example 3-2 on page 33 shows how to include this option with the JZOS batch launcher.

```
-Xscmx25m -Xshareclasses:name=MyCache
```

Tip: Type the following command under USS to view currently used caches for the shared classes:

```
ipcs -bom
```

Using log4J in the z/OS environment

The log4j API is a popular open source project (Apache Software Foundation) that provides a generic framework to include a light-weight log system in Java applications. It allows the developer to control which log statements are output

with arbitrary granularity. It is fully configurable at runtime using external configuration files. You can learn more about log4j at:

<http://logging.apache.org/log4j/docs/>

Using log4J, you can log messages at different levels including *debug*, *info*, *warn*, *error*, and *fatal*. Depending on the level of log you would like to see at execution time, you can control the output by specifying parameters in the configuration files (for example, you can specify to see only the fatal messages in the production environment, then increase it to debug level if you need to resolve issues with your application). The cost of logging using log4j is one of the highest priority items in the project. So you can expect to have the minimum impact on your application in terms of performance.

You can also direct the output of messages to different locations depending on the severity of the messages. For example, you want to see debug and info messages directed to the `System.out` stream, warn and error messages directed to the `Java.err` stream, and fatal messages to appear in the MVS console so that operators can spot the problem and take appropriate actions. You can do this by extending `org.apache.log4j.AppenderSkeleton` and using the appender in your Java application. Example 3-4 shows an example of how you can direct log messages to different output locations suitable in a z/OS environment.

Example 3-4 Example of z/OS log4j customized logger

```
package com.ibm.itso;

import org.apache.log4j.AppenderSkeleton;
import org.apache.log4j.Level;
import org.apache.log4j.spi.LoggingEvent;
import com.ibm.jzos.ZUtil;

public class ZOSAppender extends AppenderSkeleton {
    protected void append(LoggingEvent event) {
        if (event.getLevel() == Level.DEBUG
            || event.getLevel() == Level.INFO) {
            // If debug or info, then write to Java standard output stream.
            System.out.println(createMessage(event));
        } else if (event.getLevel() == Level.WARN
            || event.getLevel() == Level.ERROR) {
            // If warning or error, then write to Java error output stream.
            System.err.println(createMessage(event));
        } else if (event.getLevel() == Level.FATAL) {
            // If fatal, then write to MVS log.
            //System.err.println(createMessage(event));
        }
    }
}
```

```

        MvsConsole.wto(createMessage(event), 0x0020, 0x4000);
    } else {
        System.err.println(getName() +
            ": could not determine the message level: "
            + createMessage(event));
    }
}

/**
 * Formats the message into 'time [threadName] level
 * loggerName - message'.
 */
private String createMessage(LoggingEvent event) {
    return (event.timeStamp - LoggingEvent.getStartTime())
        + " [" + event.getThreadName() + "] "
        + event.getLevel() + " " + event.getLoggerName() + " - "
        + event.getRenderedMessage();
}

public boolean requiresLayout() {
    return false;
}

public void close() {
}
}

```

Example 3-5 illustrates how to use a customized logger that is suitable in a z/OS environment.

Example 3-5 Example of using customized logger in a z/OS environment

```

package com.ibm.itso;

import java.util.Properties;
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;

public class Log4JDemo {

    public static void main(String[] args) throws Exception {
        Logger logger = Logger.getRootLogger();

        // Properties can be set in a config file as well.
        Properties p = new Properties();
    }
}

```

```

p.setProperty("log4j.rootLogger", "DEBUG, A1");
p.setProperty("log4j.appender.A1", "com.ibm.itso.ZOSAppender");
PropertyConfigurator.configure(p);

int total = 0;
total = processNumber("1", total, logger);
total = processNumber("3", total, logger);
total = processNumber("4", total, logger);
total = processNumber("10", total, logger);
total = processNumber("1g", total, logger);
System.out.println("Total = " + total + ".");
}

/**
 * This methods accepts a number (newNumber) to be added to
 * the total. It converts the given number from String to Java int
 * to perform the calculation. It only adds the number if it is
 * less than 10. Else prints a warning message.
 */
private static int processNumber(String newNumber, int total,
    Logger logger) {
    try {
        int i = Integer.parseInt(newNumber);
        if (i < 10) {
            total += i;
            logger.info("processNumber is able to add the given number "
                + i + " to the total. Total = " + total + ".");
            return total;
        } else {
            logger.warn("processNumber is unable to add the given number "
                + i + ". Number must be less than 10.");
            return total;
        }
    } catch (Exception e) {
        logger.fatal("processNumber encounter fatal problem. Check: ["
            + newNumber + "]. It must be a deciman number less than 10.");
        logger.fatal(e);
        return total;
    }
}
}

```

3.4 Console communication with batch jobs

It is a common practice to use the Write-To-Operator (WTO) macro by running programs (jobs) to write messages to the job log and system console. The macro is typically used to report the status of the program. For instance, it is used to record events whenever failure occurs in a long-running job. Based on the messages, the operator can decide to take appropriate actions to resolve the problems. Under such circumstances or whenever the operator needs to interact with the job, it is common to use the MVS stop (P) command to stop the running job, or use the MVS modify (F) command to send arbitrary commands to change the behavior of the job. Running jobs are programmed to accept and respond to these commands. JZOS provides APIs to add these capabilities to Java applications. In this section, we present how to interact with jobs via the MVS console, and how to use these APIs in Java applications.

3.4.1 MVS console communication overview

System Display and Search Facility (SDSF) provides the interface to the system log and ways to communicate with running jobs using the MVS commands. Within SDSF, SDSF Status Display (type 'ST') displays a list of jobs currently running on the system. Several commands to filter the jobs are summarized here:

- ▶ *owner XXX**: Filters jobs by owner.
- ▶ *prefix YYY**: Filters jobs by job name prefix.
- ▶ *sj*: Displays the JCL script for the job and possibly resubmits the job.
- ▶ *s*: Displays job output produced by JES.
- ▶ *?*: Displays the job data set.

| NP | JOBNAME | JobID | Owner | Prty | Queue | C | Pos | Saff | ASys | Status | CSR |
|----|----------|----------|---------|------|-----------|---|-----|------|------|--------|-----|
| | CONSOLE | J0B06990 | KENICHI | 8 | EXECUTION | A | | | | SC76 | |
| | KENICHI | TSU06969 | KENICHI | 15 | EXECUTION | | | SC76 | | SC76 | |
| | JZ0SEXPL | J0B06964 | KENICHI | 1 | PRINT | A | 273 | | | | |
| | KENICHI | TSU06882 | KENICHI | 1 | PRINT | A | 274 | | | | |
| | JAVACOMP | J0B06983 | KENICHI | 1 | PRINT | A | 292 | | | | |
| | JAVACOMP | J0B06986 | KENICHI | 1 | PRINT | A | 293 | | | | |
| | JAVACOMP | J0B06988 | KENICHI | 1 | PRINT | A | 294 | | | | |

Figure 3-1 SDSF Status Display showing a list of jobs

Figure 3-1 shows the SDSF Status Display displaying a list of jobs currently running. From the display, various MVS commands can be issued to the running jobs, which we now describe.

MVS stop command

The MVS stop command is used to stop a currently running job. The syntax of the MVS stop command is as follows:

```
/p jobName
```

MVS modify command

The MVS modify command is used to send various commands to a currently running job. The syntax of the MVS modify command is as follows:

```
/f jobName,commands
```

This command sends the specified commands to the specified job. The job receiving the commands needs to be programmed to receive and react to the commands.

Browsing MVS system log

The MVS system log records all activities on the system and can be viewed via the SDSF SYSLOG display, which can be viewed by typing the **log** command in SDSF. Figure 3-2 displays the SDSF SYSLOG panel. Because the log includes all the information from all the jobs running on the system, it can be large. So, it is convenient to search the log backward from the bottom (the latest information) using the following commands:

```
bottom
f xxx prev
```

where **bottom** moves to the end of the log, **f** to invoke the find command, **xxx** is the string to search for and **prev** specifies to search backward.

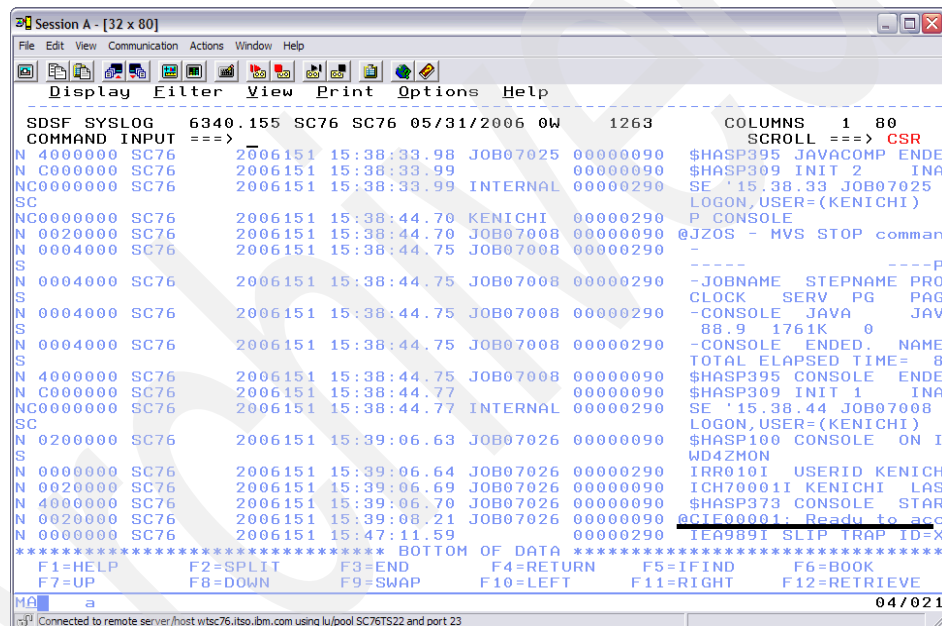


Figure 3-2 SDSF SYSLOG display

3.4.2 JZOS APIs for console communication

JZOS provides a set of APIs to implement console communication including Write-To-Operator (WTO), and the interface for the MVS command for stop (P) and modify (F). These APIs are defined in the `com.ibm.jzos.MvsConsole` class.

The method available to perform WTO:

```
public static int wto(String msg, int routeCode, int descriptor);
```

This API writes messages to the system log and return 0 if successful.

To receive commands at job startup and handle the MVS modify command, a JZOS Java application needs to register a listener using the following method:

```
public static void registerMvsCommandCallback(MvsCommandCallback
callback);
```

where `MvsCommandCallback` is a Java interface class that defines the following methods:

```
public void handleModify(String modifyCommand);
public void handleStart(String startParameters);
public void handleStop();
```

`registerMvsCommandCallback` expects to receive a reference to an object that implements the `MvsCommandCallback` interface. This is where application-specific implementation of handler is required. Whenever the job receives an MVS modify command, it invokes the `handleModify()` method of the handler registered for the application. `modifyCommand` is a string that is used to pass the commands typed by the operator. For example, if the operator types `/f JAVAJOB,go to sleep`, `modifyCommand` has the value `go to sleep`.

Note that the MVS start command will have already been issued by the time the callback is registered. Therefore, the `handleStart()` method is fired immediately upon registration. This method is used to obtain the start parameters.

An example of how to use these APIs to communicate via console is presented in the next section.

There are several other useful methods defined in the `com.ibm.jzosZUtil` class. Some of them are described here:

- ▶ `getCurrentJobname()`
Get the current MVS job name.
- ▶ `getCurrentUser()`
Get the current MVS user name.
- ▶ `getDefaultPlatformEncoding()`
Get the default platform encoding for this instance of the JVM.
- ▶ `getEnv()`
Get the value of the environmental variable (for example, `CLASSPATH`).
- ▶ `getJavaVersionInfo()`
Retrieves information about the version of Java.

3.4.3 JZOS console communication example

In this section, we present how to write a Java program which performs the console communication using JZOS APIs. The program reads a message to print from the standard input stream when it is started. Once it reads the initial message, it prints a message to the system log that the program is ready to accept commands. The program loops and prints the periodic message indefinitely. During the execution, it accepts the MVS modify command to change the message to print. Example 3-6 shows the complete program.

Example 3-6 Illustration of MVS console communication from a Java program

```
import java.io.*;
import com.ibm.jzos.*;

public class ConsoleInteractionExample {
    public static String msg = null;
    public static String msgs[] = {"Hello World!",
        "Hello from Poughkeepsie!",
        "Hello from ConsoleInteractionExample!"};
    public static String unknownMsg = "Invalid! type [0, 1, or 2]";

    public static void main(String arg[]) throws IOException {
        byte inputBuf[] = new byte[1000];
        int n;

        // Receive the initial message to print from the standard input.
        n = System.in.read(inputBuf);
        if (n > 0) {
            String enc = ZUtil.getDefaultPlatformEncoding();
            msg = new String(inputBuf, 0, n, enc);
        } else {
            msg = "No message read from the standard input";
        }
        System.out.println("Initial msg: " + msg);

        // Write to the system log that the system is ready
        // to accept commands.
        MvsConsole.wto("CIE00001: Ready to accept commands.",
            0x0020, 0x4000);

        // Test START and MODIFY command.
        MvsConsole.registerMvsCommandCallback(new MvsCommandCallback() {
            public void handleModify(String s) {
                System.out.println("Received command [" + s + "]");
                if (s.equals("0")) {

```

```

        msg = msgs[0];
    } else if (s.equals("1")) {
        msg = msgs[1];
    } else if (s.equals("2")) {
        msg = msgs[2];
    } else {
        msg = unknownMsg;
    }
}

public void handleStart(String s) {
}

public boolean handleStop() {
    return true;
}
});

// Loop forever to print the selected message.
for (int i = 0;;) {
    System.out.println(i++ + ": " + msg);
    try {
        Thread.sleep(10000);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
}

```

Notes:

- Reading from the standard input stream:
in JZOS, standard input stream is read from the STDIN DD card; an example follows:

```

//STDIN DD *
My original message from JCL
/*

```

- A WTO message is written to the system log.
- The listener for MVS commands is registered via the `MvsConsole.registerMvsCommandCallback` method.
- Standard output from the program after issuing several modify commands:
Initial msg: My original message from JCL

0: My original message from JCL
1: My original message from JCL
Received command [0]
2: Hello World!
3: Hello World!
4: Hello World!
Received command [2]
5: Hello from ConsoleInteractionExample!

Java development and job management with Eclipse

This chapter provides information about how to develop Java applications for z/OS with Eclipse tooling. This includes local Java development as well as remote deployment and debugging.

Furthermore, we introduce how to leverage the FTP protocol for submitting jobs and retrieving output.

4.1 Overview

Most Java programmers develop their Java applications locally in integrated development environments (IDE) like Eclipse. For standalone Java applications on z/OS, this is more difficult because the IDE needs interaction with the host site for deploying the application, as well as for getting results.

But if you integrate your Eclipse IDE successfully into the z/OS Batch subsystem, you can achieve higher productivity for your AD staff.

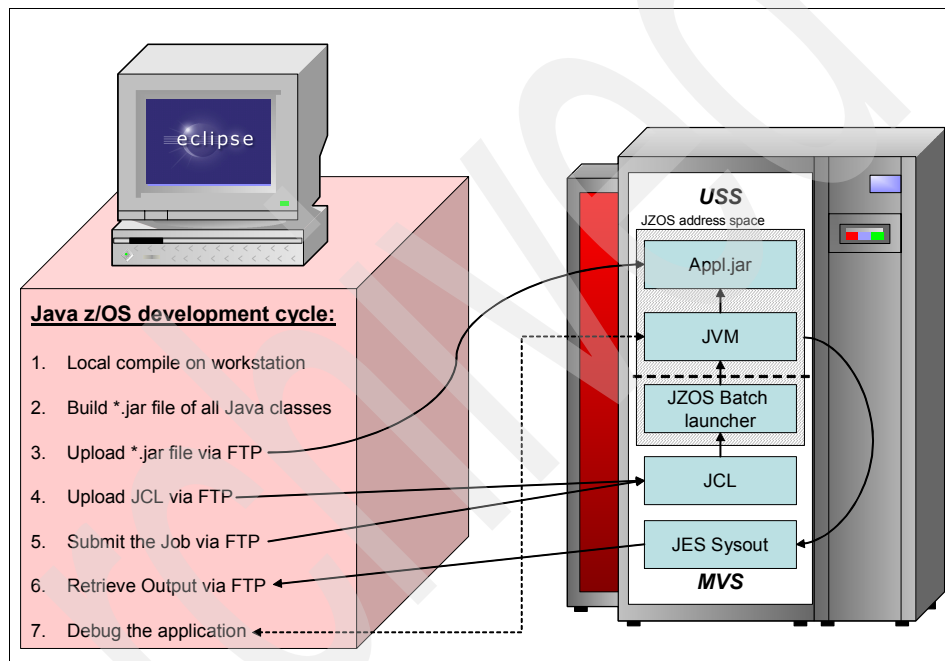


Figure 4-1 Java development for z/OS with Eclipse

Figure 4-1 illustrates how we used Eclipse for developing and debugging z/OS Java applications.

With the help of the Apache Ant build tool as described in *Java Stand-alone Applications on z/OS, Volume I*, SG24-7177, it is possible to establish the following completely automated Java z/OS development cycle in Eclipse:

1. Local development and compile of the whole Java application including the usage of z/OS-specific APIs.

2. After having compiled all *.java files, all built *.class files are packaged into one *.jar file.
3. The new built *.jar file is uploaded via FTP into Unix System Services.
4. To submit the uploaded Java application as a batch Job with JZOS, a locally—in Eclipse—developed JCL is uploaded to MVS.
5. The uploaded batch job is submitted via the FTP protocol and invokes the uploaded Java application in the JVM with the help of the JZOS launcher. Batch launcher and the Java application in the JVM are running in the same address space.
6. The JES output of the Java batch job is retrieved via FTP to the Eclipse command console.
7. If necessary, the application can be debugged in Eclipse with the help of remote debugging through the z/OS JVM.

The following sections explain how this development cycle can be fully automated with the help of Apache Ant.

4.2 Setting up Eclipse for host interaction

The Eclipse-based Java z/OS application development cycle is based on Apache Ant support. See:

<http://ant.apache.org/>

for more details.

The Eclipse Ant support allows you to create and run Ant buildfiles (some kind of scripts for Eclipse) from the workbench. These Ant buildfiles can operate on resources in the file system as well as resources in the workspace.

Output from an Ant Buildfile is displayed in the console view in the same hierarchical format seen when running Ant from the command line. Ant tasks (for example "[mkdir]") are hyperlinked to the associated Ant Buildfile, and *javac* error reports are hyperlinked to the associated Java source file and line number.

This concept allows you to easily build scripts to compile and deploy Java standalone applications for z/OS with the JZOS toolkit.

4.2.1 Prerequisites

The following prerequisites exist for this environment:

- JZOS installed properly

- ▶ Eclipse 3.1 including Java SDK installed on the workstation. See:
<http://www.eclipse.org/>
for more details.
- ▶ The Java SDK is installed properly on the host. We used SDK 5.0 in this example.

4.2.2 Set up Ant FTP support in Eclipse

To facilitate the application deployment from the Eclipse workbench running on the workstation to z/OS, the Ant FTP support in Eclipse needs to be set up.

Download commons-net-1.4.1.zip and jakarta-oro-2.0.8.zip from:

http://jakarta.apache.org/site/downloads/downloads_commons-net.cgi
http://jakarta.apache.org/site/downloads/downloads_oro.cgi

Extract commons-net-1.4.1.jar and jakarta-oro-2.0.8.jar from the zip files and put them into a directory called c:\ant_ftp_jars.

Open Eclipse and select **Window** → **Preferences** → **Ant** → **Runtime**. Under the classpath tab, select **Ant Home Entries** and click **Add External JARs**. Pick those two jar files in the directory c:\ant_ftp_jars from the file selection dialog. Then click **Open**. The Ant FTP support jars have been added to the Ant classpath.

Now the Ant FTP can be used in a buildfile running in the Eclipse workbench.

4.2.3 Import sample project

We provide a sample project that illustrates how to use Eclipse for Java development. Refer to Appendix A, “Additional material” on page 147, for details.

1. Open Eclipse and select **File** → **Import** → **Existing Projects into Workspace** → **Next**.
2. Select **Select archive file** as shown in Figure 4-2. Browse for our sample project, JZOS_BatchSample.zip, and click **Finish**.

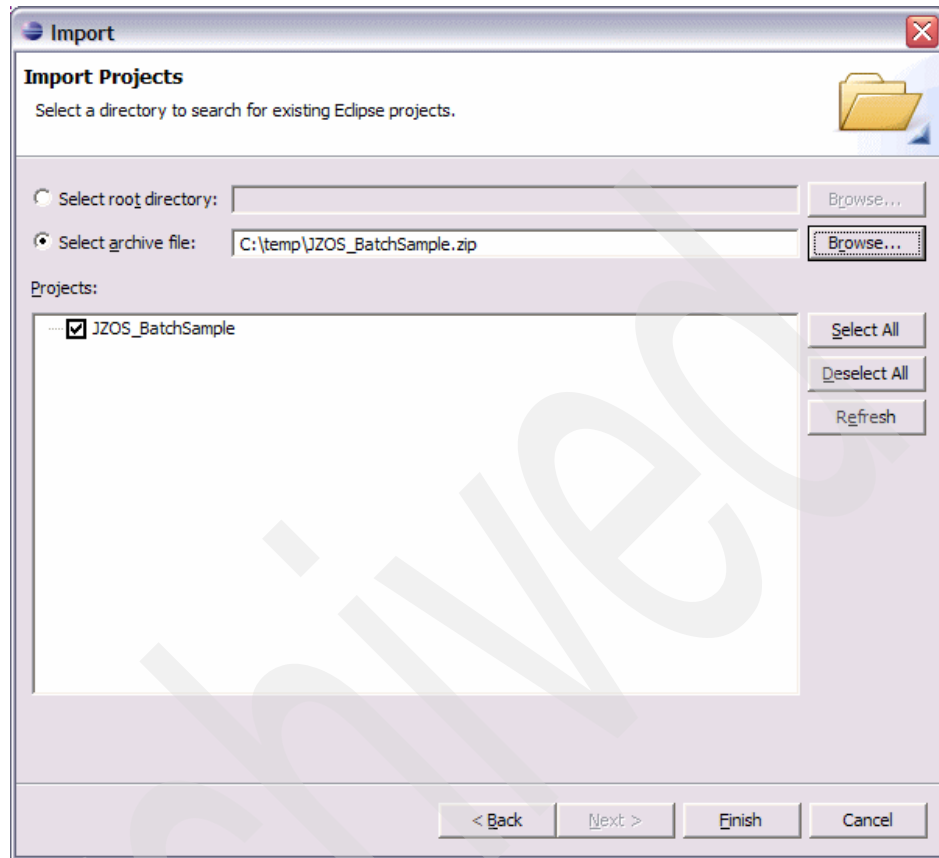


Figure 4-2 Importing Eclipse sample project

Now you should see the following project in your Package Explorer:

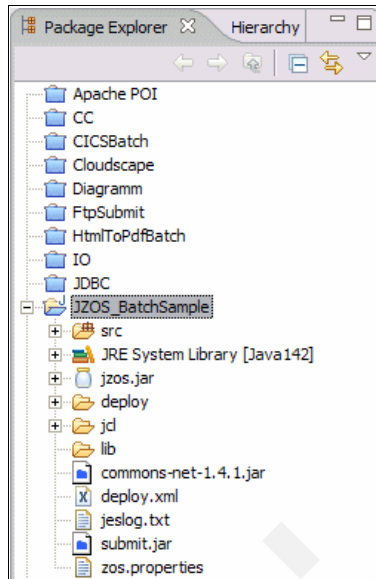


Figure 4-3 Eclipse Project Explorer

4.2.4 Configure the project

Before you can work with the project, you have to change some properties.

First, you have to tailor the JCL that submits the job. Open the JCL in the jcl folder in the Package Explorer trough by double-clicking it, as shown in Figure 4-4.

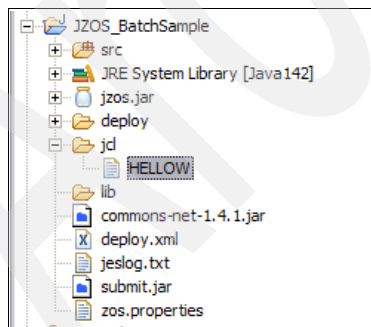


Figure 4-4 Opening JCL in the Package Explorer

The following JCL opens; see Example 4-1 on page 55.

```
//USER02Z JOB
//PROCLIB JCLLIB ORDER=JBATCH.SAMPLES.JCL
//*
/*****
/*
/* Batch job to run sample batch programs with Java5.0 VM
/*
/* Tailor the proc and job for your installation:
/* 1.) Modify the Job card per your installation's requirements
/*    The job name must match your userid plus one character!
/*    Otherwise you cannot submit the job via FTP.
/* 2.) Modify the PROCLIB card to point to your JZOS proclib
/* 3.) Modify APPL_HOME to point to the hfs directory where the
/*    application was deployed
/* 3.) Modify JZOS_HOME to point to the hfs directory where JZOS
/*    was installed
/* 4.) edit JAVA_HOME to point the location of the 5.0 JDK
/* 5.) Modify the CLASSPATH as required to point to your Java code
/* 6.) Modify JAVACLS and ARGS to launch desired Java class
/* 7.) Customize any DD statements required for the application
/*
/*****
/*
//HOLD OUTPUT JESDS=ALL,DEFAULT=Y,OUTDISP=(HOLD,HOLD)
//JAVA EXEC PROC=EXJZOSVM,VERSION='50',
// JAVACLS='com.ibm.jzos.sample.HelloWorld'
//OUTPUT DD SYSOUT=*
//STDENV DD *
# This is a shell script which configures
# any environment variables for the Java JVM.
# Variables must be exported to be seen by the launcher.

. /etc/profile
export JZOS_HOME=/jbatch/local/jzos
export APPL_HOME=/jbatch/java/sample
export JAVA_HOME=/usr/lpp/java/java50/J5.0

export PATH="$PATH":"${JAVA_HOME}"/bin:

LIBPATH="$LIBPATH":"${JAVA_HOME}"/bin
LIBPATH="$LIBPATH":"${JAVA_HOME}"/bin/classic
LIBPATH="$LIBPATH":"${JZOS_HOME}"
export LIBPATH="$LIBPATH":
```

```

# Customize your CLASSPATH here
# Add application home directory and jars to CLASSPATH
CLASSPATH=${APPL_HOME}
for i in "${APPL_HOME}"/*.jar; do
    CLASSPATH="$CLASSPATH": "$i"
done

# Add JZOS required jars to end of CLASSPATH
for i in "${JZOS_HOME}"/*.jar; do
    CLASSPATH="$CLASSPATH": "$i"
done
export CLASSPATH="$CLASSPATH":

# Set JZOS specific options
# Use this variable to specify encoding for DD STDOUT and STDERR
#export JZOS_OUTPUT_ENCODING=Cp1047
# Use this variable to prevent JZOS from handling MVS operator commands
#export JZOS_ENABLE_MVS_COMMANDS=false
# Use this variable to supply additional arguments to main
#export JZOS_MAIN_ARGS=""

# Configure JVM options
IJO="-Xms16m -Xmx128m"
# Uncomment the following line if you want to debug the application
#IJO="$IJO -Xdebug -Xrunjdpw:transport=dt_socket,server=y,address=8000"
IJO="$IJO -Djzos.home=${JZOS_HOME}"
# Uncomment the following if you want to run without JIT
#IJO="$IJO -Djava.compiler=NONE"
# Uncomment the following if you want to run with Ascii file encoding..
#IJO="$IJO -Dfile.encoding=ISO8859-1"
export IBM_JAVA_OPTIONS="$IJO "

export JAVA_DUMP_HEAP=false
export JAVA_PROPAGATE=NO
export IBM_JAVA_ZOS_TDUMP=NO
//

```

Tailor this JCL for your personal use:

1. Modify the Job card to reflect your system requirements.

Note: The job name must match your user ID plus one character!
Otherwise you cannot submit the job via FTP.

2. Modify the PROCLIB card to point to your JZOS PROCLIB.
3. Modify APPL_HOME to point to the hfs directory where you plan to deploy your application. This directory must exist on the host, of course.
4. Modify JZOS_HOME to point to the hfs directory where JZOS was installed.
5. Modify JAVA_HOME to point to the location of the 5.0 JDK™.
Remember to set the right local compiler compliance level (see 4.4.1, “Setting options” on page 67).

After having tailored your JCL, open `zos.properties` in the Package Explorer:

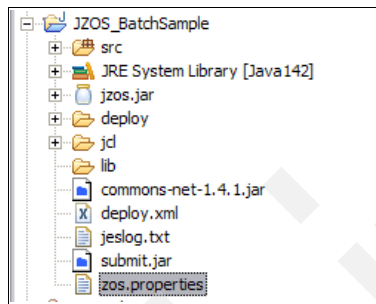


Figure 4-5 Opening `zos.properties` in the Package Explorer

The following file opens:

Example 4-2 The `zos.properties` file

```
# Customize this file or move a customized copy of this file up to
# either the Eclipse workspace directory or your "home directory"
server = myserver.ibm.com
userid = userid
password = password

# The JCL PDS dataset, which must be in single quotes:
jcl.dsn = 'JBATCH.SAMPLES.JCL'

# the home directory for deploying the application jar
appl.home = /jbatch/java/sample

# jar name
jarname = test.jar

# local JCL pathes
jclPath = jcl/HELLOW
```

```
# time to wait for output in seconds
waittime = 0

# remote debugging
debug= no
```

Tailor this properties file for your own purposes:

- ▶ Change server, userid and password.
- ▶ Modify the PDS data set to point to the one where you want to have your JCL uploaded.
- ▶ Enter a name for the *.jar file that will be uploaded to the host.
- ▶ For each Eclipse project, the JCL file names should be different to avoid overwriting members with the same name on the host. For that reason, you can change the path in the zos.properties.
- ▶ If your application runs very long, specify the time you expect it to run. See 4.3.1, “Submitting jobs and controlling output” on page 60 for more details.
- ▶ If you want to run your application in remote debug mode, turn this flag on (see 4.4, “Debugging” on page 67).

The JCL and zos.properties are read by the delopy.xml file. This Ant buildfile (see Example 4-3 on page 59) invokes all necessary steps of the Java z/OS development cycle:

1. Local compile on workstation.
This is done by the target **compile**.
2. Build *.jar file of all Java classes.
This is done by the target **buildJar**.
3. Upload *.jar file via FTP.
This is done by the target **deployJar**.
4. Upload JCL via FTP.
This is done by the target **deployJcl**.

Note: JCLs have to be uploaded in text mode. If you are experiencing problems with the encoding on the host, try the following:
Create a &USER.FTPD.RC data set with the following content:

```
SITE SBDATACONN=(IBM-1047,ISO8859-1)
```

5. Submit the Job via FTP.
This is done by the target **submit**. Here, the Ant task java invokes a little Java

application that is contained in the submit.jar file. This Java application performs submitting jobs via FTP and also retrieves the JES output.

6. Retrieve output via FTP.
See 5.)
7. Debug the application.
For debugging, see 4.4, “Debugging” on page 67.

Example 4-3 The deploy.xml file

```
<project name="deploy" default="all">
  <property file="${user.home}/zos.properties"/>
  <property file="../zos.properties" />
  <property file="./zos.properties" />

  <target name="all" depends="compile, deployJar, deployJcl, submit"
/>

  <!-- for compiling if not done automatically by IDE (Eclipse) -->
  <target name="compile">
    <mkdir dir="bin"/>
    <javac destdir="bin"
      classpath="lib/jzos.jar"
      debug="on">
      <src path="src"/>
    </javac>
  </target>

  <target name="buildJar">
    <mkdir dir="deploy"/>
    <jar destfile="deploy/${jarname}">
      <fileset dir="bin">
        <include name="**/*"/>
      </fileset>
    </jar>
  </target>

  <target name="deployJar" depends="buildJar" >
    <echo message="Copying files to ${server}:${appl.home}..." />
    <ftp server="${server}"
      userid="${userid}"
      password="${password}"
      remotedir="${appl.home}"
      depends="no"
      binary="yes"
      umask="002">
```

```

        verbose="yes" >
        <fileset dir="deploy" includes="" casesensitive="yes" />
    </ftp>
</target>

<target name="deployJcl" >
    <echo message="Copying JCL members to ${server}:${jcl.dsn}..." />
    <ftp server="${server}"
        remotedir="${jcl.dsn}"
        userid="${userid}"
        password="${password}"
        binary="no"
        verbose="yes"
        depends="no" >
        <fileset dir="jcl" >
            <include name="*" />
        </fileset>
    </ftp>
</target>

<target name="submit">
    <java jar="submit.jar"
        fork="true">
        <arg line="${server} ${userid} ${password} ${jclPath} ${debug}
${waittime}" />
    </java>
</target>
</project>

```

4.3 Developing Java applications

In the following sections we explain how to develop a Java batch application using JZOS.

4.3.1 Submitting jobs and controlling output

After having customized the sample project, you can deploy it. Right-click on `delploy.xml` and select **Run As** → **Ant Build...** as shown in Figure 4-6.

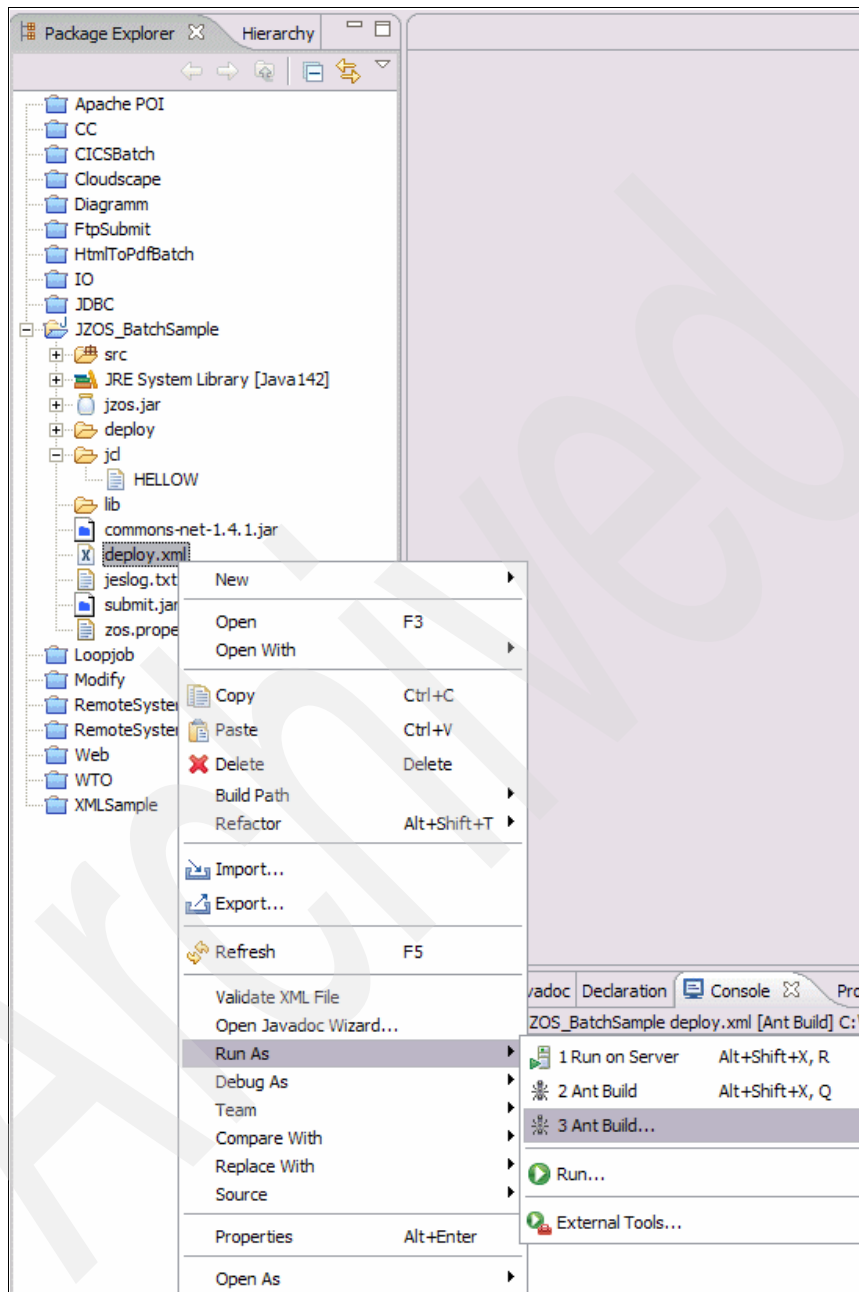


Figure 4-6 Deploying the Java application

The window shown in Figure 4-7 on page 62 appears.

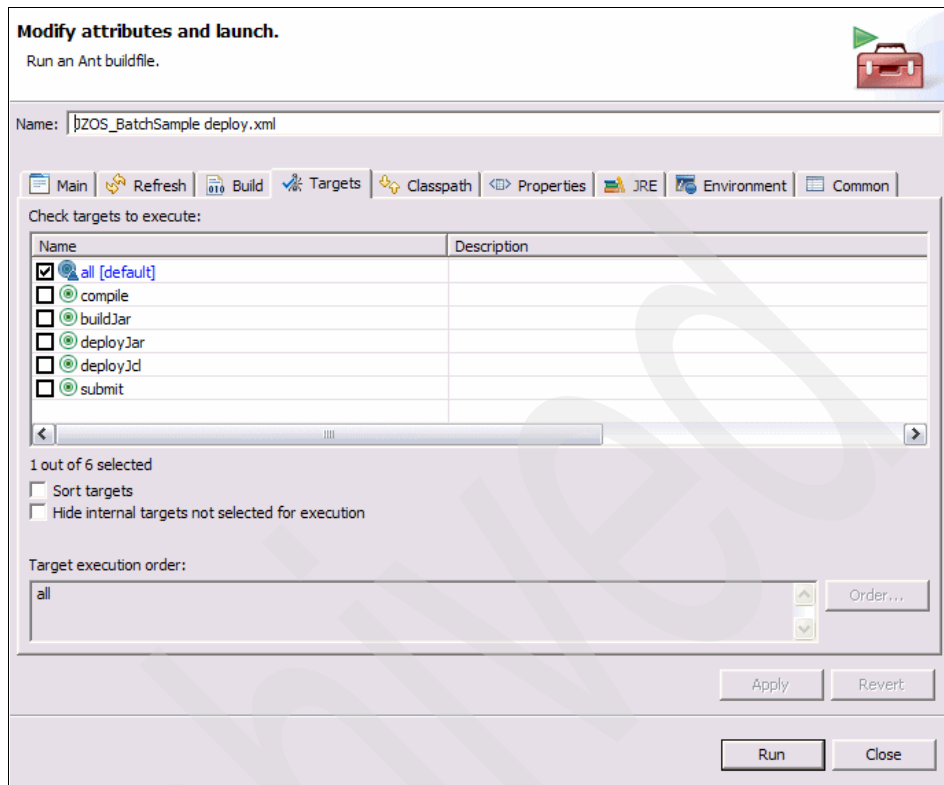
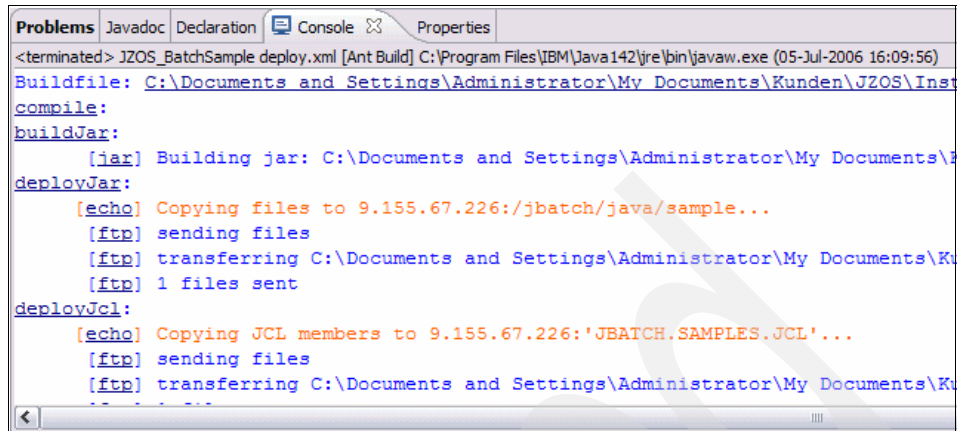


Figure 4-7 Ant build options

Ensure that **all** is selected and select **Run**.

In the Eclipse console, you can now see the status of each Ant build step (see Figure 4-8 on page 63).



```
<terminated> JZOS_BatchSample deploy.xml [Ant Build] C:\Program Files\IBM\Java142\jre\bin\javaw.exe (05-Jul-2006 16:09:56)
Buildfile: C:\Documents and Settings\Administrator\My Documents\Kunden\JZOS\Inst
compile:
buildJar:
    [jar] Building jar: C:\Documents and Settings\Administrator\My Documents\Kunden\JZOS\Inst\BatchSample.jar
deployJar:
    [echo] Copying files to 9.155.67.226:/jbatch/java/sample...
    [ftp] sending files
    [ftp] transferring C:\Documents and Settings\Administrator\My Documents\Kunden\JZOS\Inst\BatchSample.jar to 9.155.67.226:/jbatch/java/sample...
    [ftp] 1 files sent
deployJcl:
    [echo] Copying JCL members to 9.155.67.226:'JBATCH.SAMPLES.JCL'...
    [ftp] sending files
    [ftp] transferring C:\Documents and Settings\Administrator\My Documents\Kunden\JZOS\Inst\BatchSample.jcl to 9.155.67.226:'JBATCH.SAMPLES.JCL'...
```

Figure 4-8 Eclipse console output

After compiling and uploading all necessary files to the host, the job is submitted via FTP and the JES SYSOUT is displayed on the Eclipse console. You should see a “HelloWorld” and an error message from us.

Finally, the job is purged.

Note: The output control is based on FTP. Therefore, the job’s SYSOUT must be placed in the HOLD queue (see JCL in Example 4-1 on page 55). For that reason, you have to wait for the output until the job has finished.

Sometimes, you might have a very long-running job. In that case, you have to specify a wait time in the `zos.properties` file. This is the time that Eclipse waits for getting the output after the job was submitted. Otherwise, you would get a time-out and no output would be displayed.

4.3.2 Customizing the Java application

The last sections described our sample HelloWorld application. In this section, we describe the necessary steps for how to develop your own Java standalone application for z/OS with Eclipse.

We recommend to take the HelloWorld Eclipse sample project as template for new Java projects. For that reason, we suggest to copy the whole working sample project into a new project. To do this, right-click the project in the Package Explorer and select **Copy**. Press Ctrl+v and enter a name for the new project.

After having done this, you can customize the project for your own purposes. In the Package Explorer, open the src folder:

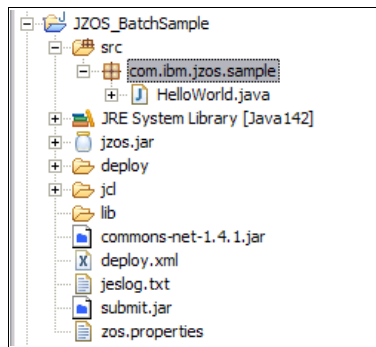


Figure 4-9 The src folder containing all Java sources

This folder contains all Java sources. Into it you can put your own packages and Java files.

If you have additional packages in *.jar files, also put these into the deploy folder because they have to be uploaded to z/OS, too. Because of the script in the JCL, those *.jar files should be included automatically in the classpath.

If you have other classes that need to be included in the classpath, specify them in the JCL.

To realize local compiling with Eclipse, all necessary *.jar files have to be included in the project's Java classpath, too. Right-click the project and select **Build Path** → **Configure Build Path...** as shown in Figure 4-10 on page 65.

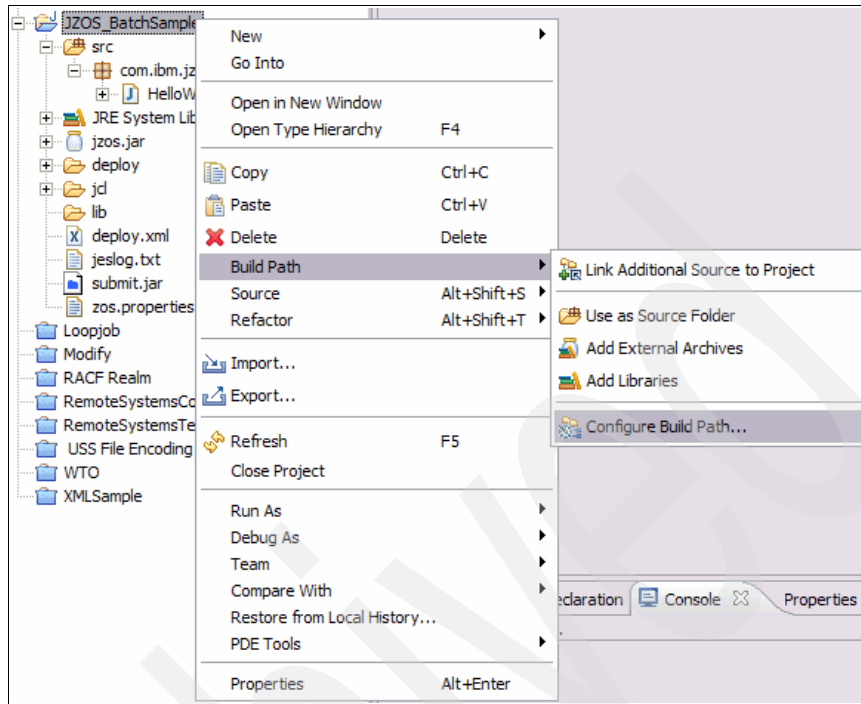


Figure 4-10 Changing application's build path

Add your *.jar files that you have put in the deploy folder to your classpath by selecting **Add JARs...** as shown Figure 4-11 on page 66.

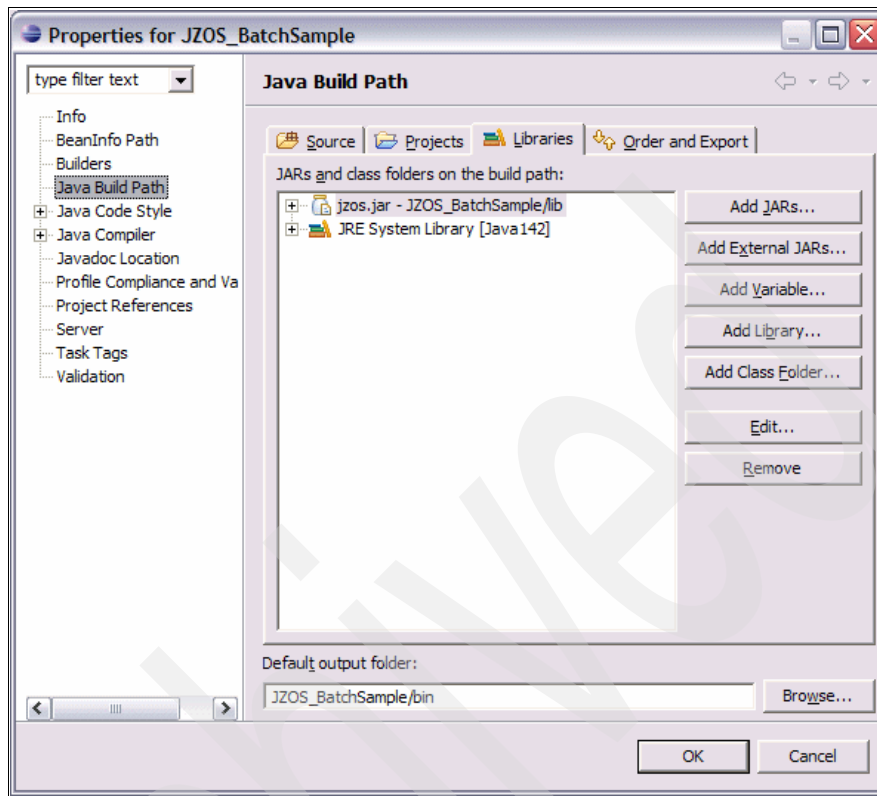


Figure 4-11 Customizing classpath

The next step for customizing your Java application is to modify JAVACLS and ARGS in the JCL to launch the desired Java class.

After that, you should customize any DD statements required for the application in the JCL.

If you have modified the name of the jarname in the zos.properties, delete the original file in the deploy folder before your first deployment to avoid uploading both *.jar files to the host.

Finally, we recommend to change the JCL's name in the jcl folder after having copied the whole project, because a deployment of the new project would cause an overwriting of the JCL from the previous project on the host. Therefore, you also have to modify the jclPath in the zos.properties to contain the new JCL's name. Furthermore, appl.home in zos.properties should be changed for a new project to achieve a clear separation from other Java projects.

4.4 Debugging

When developing complex applications, it is usually necessary to perform some debugging. The following sections explain how to remotely debug the Java application.

4.4.1 Setting options

If you want to debug your application with Eclipse, make sure that the Java class file generation options for debugging have been chosen in the Eclipse Preferences dialog. From the Window menu, click **Preferences**. The Preferences dialog shows up, as shown in Figure 4-12.

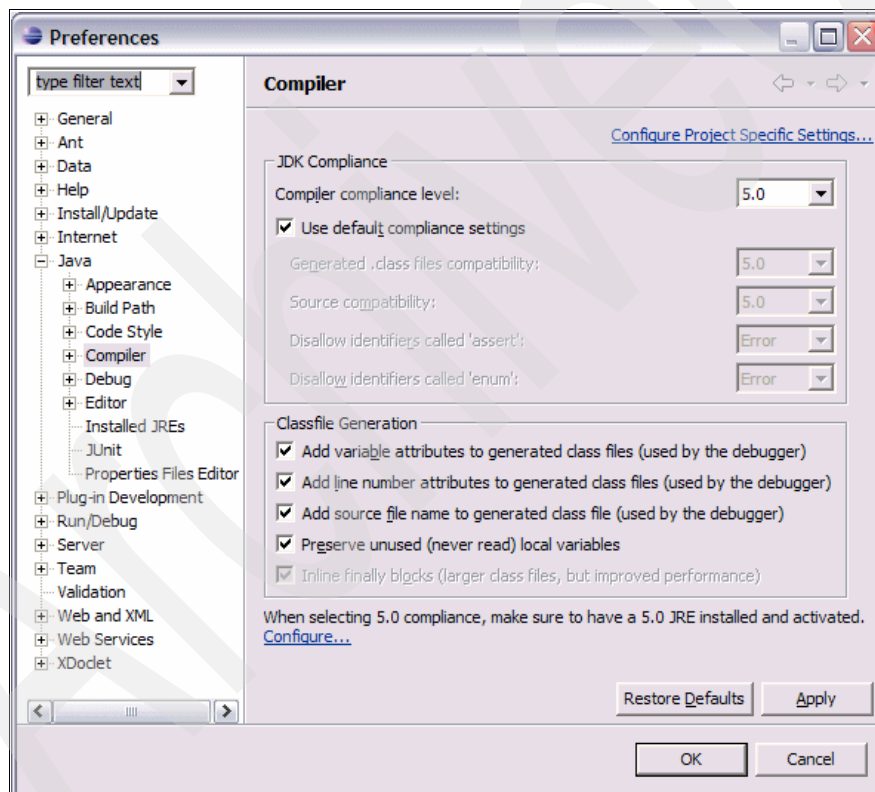


Figure 4-12 Eclipse Preferences dialog

Under **Java** → **Compiler**, the first three Classfile Generation options should be checked. Also verify that the right compiler compliance level is set. We used 5.0

in our scenario. Rebuild the project. Now the program contains all necessary information for debugging.

To enable the JVM on the host for debugging, JVM options have to be set. Open the JCL in the jcl folder and uncomment the debugging line as shown in Example 4-4.

Example 4-4 JVM debugging options in the JCL

```
# Uncomment the following line if you want to debug the application
IJO="$IJO -Xdebug -Xrunjdw:transport=dt_socket,server=y,address=8000"
```

-Xdebug and -Xrunjdw are the JVM options that support remote debugging. The transport=dt_socket specifies the communication transport mechanism between the JVM (server) and debugger (client). Don't forget to set server=y. It is the option that lets the JVM listen for the debugger on the port specified in the address option. Otherwise, the JVM tries to connect to a nonexistent server because the default value of the server option is n. In the above shell script, the JVM starts as the debugging server that listens on port 8000.

Other options and explanations for -Xrunjdw can be found in Table 4-1.

Table 4-1 -Xrunjdw options

| Option Name and Value | Description | Default |
|--------------------------|-------------------------------|---------|
| suspend=y n | Wait on startup? | y |
| transport=<name> | Transport spec | none |
| address=<listen/attach | Address> transport spec | "" |
| server=y n | Listen for debugger? | n |
| launch=<command line> | Run debugger on event | none |
| onthrow=<exception name> | Debug on throw | none |
| onuncaught=y n | Debug on any uncaught? | n |
| strict=y n | Skip JVMDI bug workarounds? | n |
| stdalloc=y | IN Use C Runtime malloc/free? | n |

Before you can deploy the application for debugging, debug = yes has to be set in the zos.properties file. This option forces the submit plug-in not to wait for

output in the command line. Otherwise you probably would get a time-out as described in 4.3.1, “Submitting jobs and controlling output” on page 60.

Note: If debug = yes is set in the zos.properties file, the job will not be purged automatically from the plug-in. Unfortunately, you have to do this manually in SDSF in this case.

An alternative to having the Job automatically purged by Eclipse is to set debug = no in the zos.properties file and set the waittime so that it is sufficient for your debugging work. This method will also display the JES SYSOUT in the command line, but only at the end, of course. The main disadvantage is that your debugging work completely stops after the waittime and you might not have finished with your debugging activities there.

4.4.2 Working with the Eclipse debugger

After having set all necessary options, you can deploy the application as described in 4.3.1, “Submitting jobs and controlling output” on page 60

Now you are ready to use the Eclipse debugger. For more information on how to use the Eclipse debugger in the workbench, refer to *Java Stand-alone Applications on z/OS, Volume I*, SG24-7177, section 5.3.5.

Using WebSphere Developer for zSeries

In this chapter we present an overview of WebSphere Developer for zSeries (WDz), including the software and hardware requirements for workstation and host. We also present the general usage scenarios of WDz features and the remote debugging capability of WDz. Further information about the product can be found at:

<http://www-306.ibm.com/software/awdtools/devzseries/>

5.1 Overview of WDz

WebSphere Developer for zSeries (WDz) provides the most comprehensive tool set for enterprise application development in IBM's Eclipse-based software development tools portfolio. WDz provides a set of tools for developing applications for System z, which makes traditional mainframe development, Web development, and integrated mixed workload or composite development faster and more efficient. Some of the capabilities of WDz are summarized as follows:

- ▶ Development of ASM, COBOL and PL/I applications and services deployed to CICS, IMS, batch, and DB2 stored procedure-based environments
- ▶ An interactive, workstation-based environment with quick access to IBM z/OS data sets, zFS, and JESS
- ▶ Enterprise Generation Language (EGL) for z/OS
- ▶ Service Flow Modeler (SFM) for integrating CICS applications
- ▶ Debugging of COBOL, PL/I and assembler applications
- ▶ All of Java/J2EE and Web related tools from Eclipse, IBM Rational® Application Developer for WebSphere Software (RAD), and IBM Rational Web Developer for WebSphere Software (RWD)

Figure 5-1 on page 73 shows a typical view of a WDz z/OS project perspective¹. The presentation of resources on z/OS gives a view as if you are working with local resources just like any other projects in the Eclipse environment. With an integrated editor, Ipex, most of the desired features such as code assist for COBOL, PL/I, HLASM and JCL, and syntax checks are provided. For further details about the WDz product, visit:

<http://www.ibm.com/software/awdtools/devzseries/>

¹ Perspective is an Eclipse concept which organizes a set of views for a particular activity. It is highly configurable and gives a multi-dimensional feel to the user. Examples of perspectives include Java, J2EE, Debug, and z/OS perspectives.

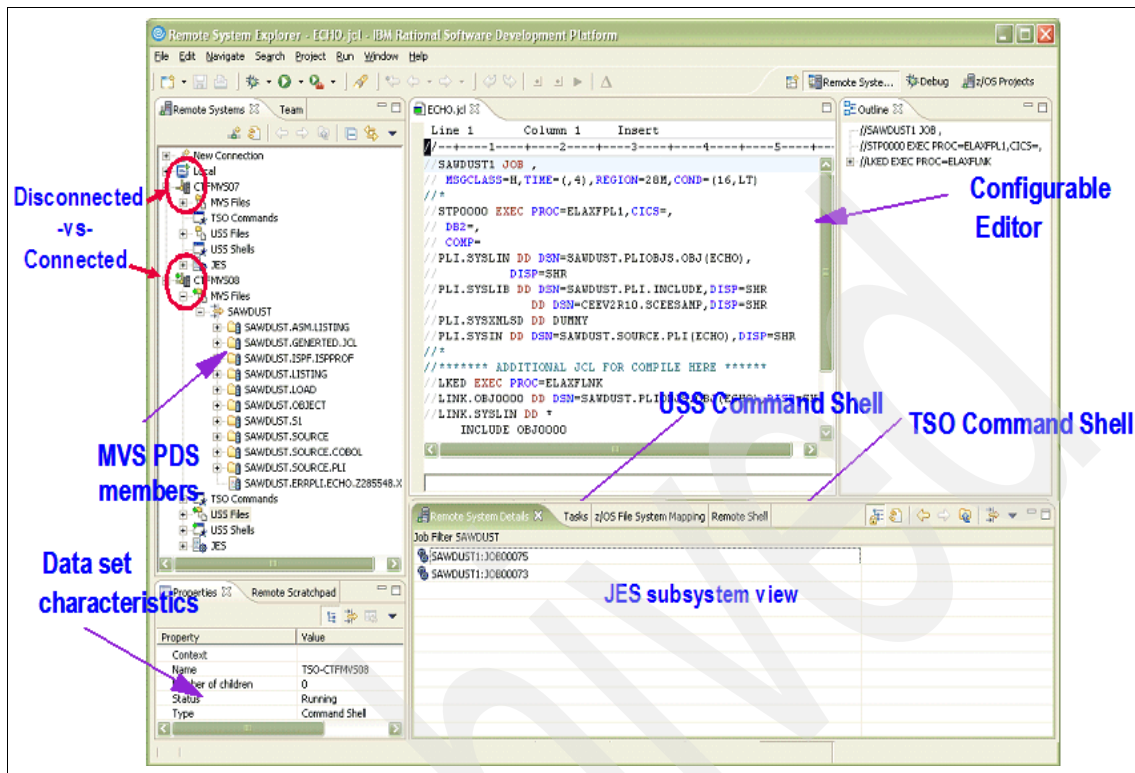


Figure 5-1 Typical view of WDz workspace

Figure 5-2 illustrates an overview of the WDz client architecture. Note that all WDz tools are built on top of IBM Eclipse Development Kit, RWD and RAD. WDz delivers a nice end-to-end tool set for enterprise application development for System z while leveraging Java/J2EE tools.

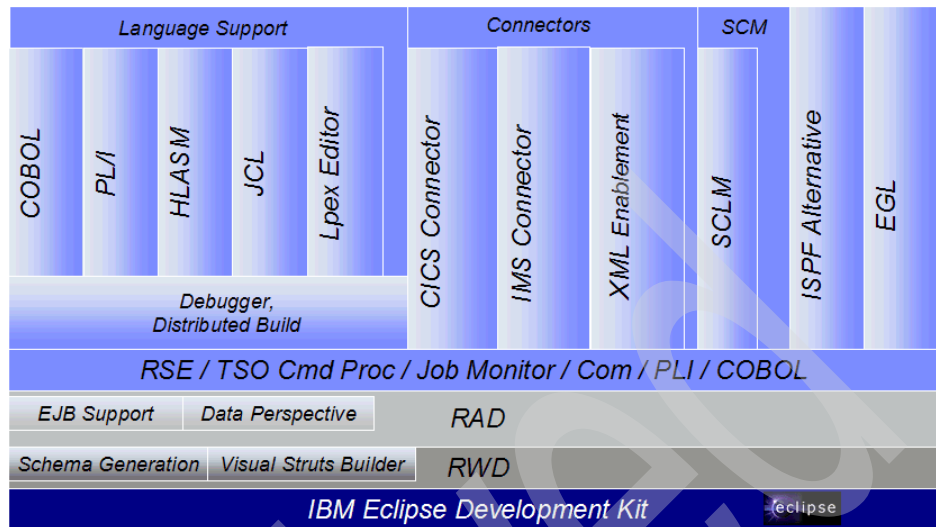


Figure 5-2 WDz client architecture

Workstation requirements

- ▶ Intel® Pentium® III 800 MHz processor; higher is recommended.
- ▶ 768 MB RAM, 1 GB or higher is recommended.
- ▶ VGA display of 1024x768 minimum required.
- ▶ 1.92 ~ 3.0 GB disk space required.
- ▶ Windows XP Professional with Service Pack 1 or later.
Or Windows 2000 Professional, Server, or Advanced Server with Service pack 3 or later.
Or Windows 2003 Standard or Enterprise.
- ▶ TCP/IP installed and configured.

Figure 5-3 illustrates an overview of the WDz host architecture.

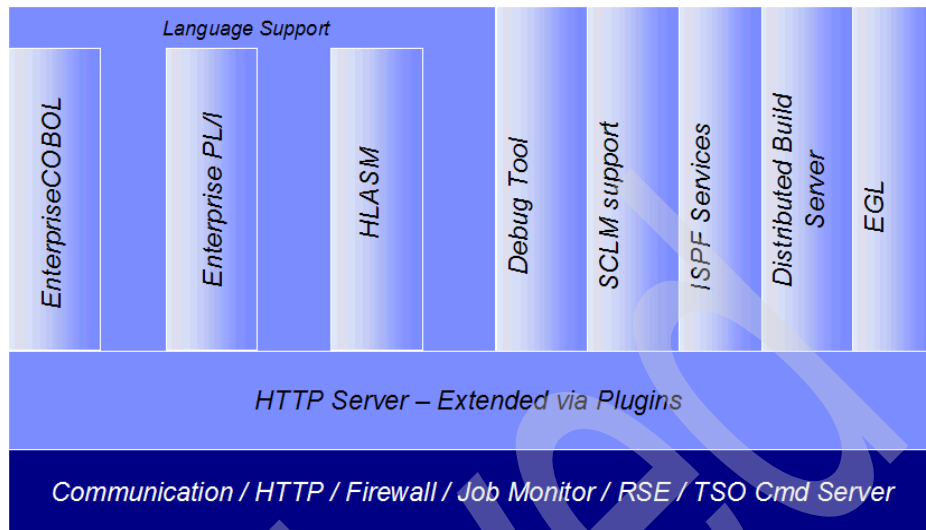


Figure 5-3 WDz host architecture

Host (z/OS) requirements

- ▶ z/OS V1.4 or higher.
- ▶ COBOL compiler - one of the following to compile COBOL programs:
 - IBM COBOL for OS/390 and VM V2.1 (5648-A25) plus PTFs
 - IBM Enterprise COBOL for z/OS V3.1 (5655-G53) plus PTFs
 - IBM Enterprise COBOL for z/OS V3.2, or later (5655-G53)
- ▶ EGL for COBOL runtime support for z/OS - support the EGL for COBOL runtime support for z/OS:
 - IBM Enterprise Developer Server for z/OS V5.0 (5655-I57) plus PTFs
- ▶ PL/I compiler - one of the following to compile PL/I programs:
 - IBM PL/I for MVS and VM V1.1 (5688-235)
 - IBM VisualAge® PL/I for OS/390 V2.2 (5655-B22)
 - IBM Enterprise PL/I for z/OS V3.1, or later (5655-H31) plus PTFs
- ▶ Debug tool - one of the following to support remote debugging of COBOL and PL/I programs:
 - IBM Debug Tool for z/OS and OS/390 V3.1 (5655-H32) plus PTFs
 - IBM Debug Tool for z/OS V4.1 (5655-L24) plus PTFs
 - IBM Debug Tool for z/OS V5.1, or later (5655-M18), plus PTFs

- IBM Debug Tool for z/OS V6.1, or later (5655-P14), plus PTFs
- ▶ CICS transaction server - one of the following to support applications with embedded CICS statements:
 - IBM CICS Transaction Server for OS/390 V1.3 (5655-147)
 - IBM CICS Transaction Server for z/OS V2.2, or later (5697-E93)
 - IBM CICS Transaction Server for z/OS V3.1 (5655-M15)
- ▶ Service Flow Modeler
 - The CICS Service Flow Feature of IBM CICS Transaction Server for z/OS V3.1 (5655-M15)
- ▶ IMS/ESA® - one of the following must be installed to support applications using the IMS database and data communications:
 - IMS/ESA V7.1, or later (5655-B01)
 - IMS/ESA V8.1, or later (5655-C56)
 - IMS/ESA V9.1, or later (5655-J38)
- ▶ IBM SDK for z/OS Java 2 Technology Edition: the following must be installed to support applications using RSE server:
 - IBM SDK for z/OS Java 2 Technology Edition, V1.4 (5655-I56)
- ▶ DB2 UDB - one of the following must be installed to support DB2 UDB for z/OS or OS/390:
 - DB2 UDB V6.1, or later (5645-DB2)
 - DB2 UDB V7.1, or later (5675-DB2), plus PTFs
 - DB2 UDB V8.1, or later (5625-DB2)
- ▶ Software Configuration and Library Manager (SCLM)
 - REXX/370 Library, or Alternate Library, V1.3 (5695-014)

For each of the following functions, required FMIDs need to be installed. Refer to *IBM WebSphere Developer for zSeries Host Planning Guide*, SC31-6599-03 for more details.

Table 5-1 Required FMIDs for the WDz function

| | |
|--|---------------------------|
| WebSphere Developer for zSeries Host Connectivity | H001600, H002600 |
| WebSphere Developer for zSeries JES Connectivity | H001600, H002600 |
| WebSphere Developer for zSeries RSE Remote Compile | HEDS500, H001600, H002600 |

| | |
|---|---------------------------|
| WebSphere Developer for zSeries Host Connectivity | H001600, H002600 |
| SCLM Access | HEDS500, H001600, H002600 |
| Common SCM Access | H001600, HCMA601 |
| Error Feedback | HEDS500, H001600, H002600 |
| Remote Debugging | HEDS500, H001600, H002600 |
| DB2 Stored Procedures | HEDS500, H001600, H002600 |
| EGL | HEDS500, H284500 |
| BIDI | HBDI601 |

5.2 Setting up a basic connection with the host

Before starting to use the many tools available in WDz, a connection needs to be established correctly in order to facilitate communication between the host and workstation. In order to set up a connection, you need to have a host name (TCP/IP address), JES job monitor port number, and port number for the remote daemon to manipulate MVS data sets and USS files. Port numbers are installation-dependent, so check with the system programmer who installed the host components (the default port numbers are 6715 and 4035 respectively). If the connection is established correctly, you will be prompted for your user name and password.

To establish the connection with a host, complete the following steps:

1. Open the z/OS projects perspective: **Window** → **Open Perspectives** → **Other** → **z/OS Projects** (default perspective in WDz).
2. In the Remote Systems view, right-click and select **New Connection** from the context menu.
3. Specify the information described above in the Remote System Connection wizard.

Once the connection is established correctly, you should see an entry for the connection in the Remote Systems view. Figure 5-4 illustrates a sample view showing two connections to different hosts.

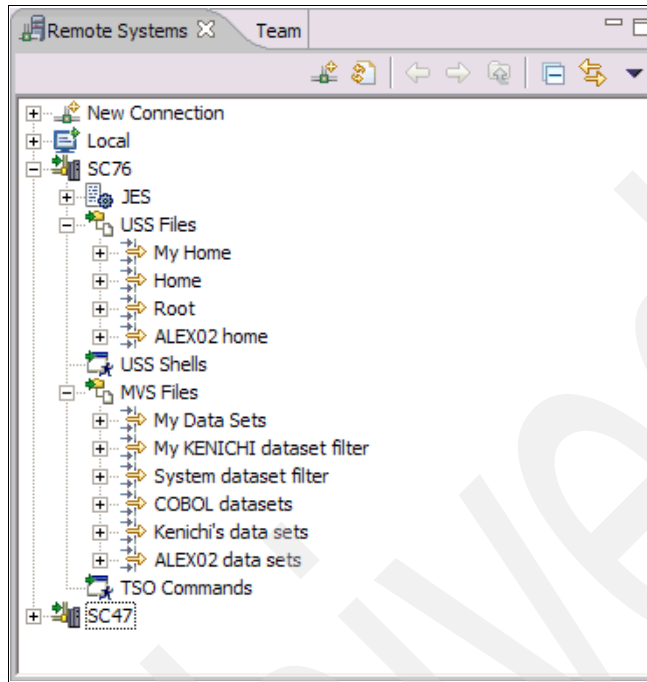


Figure 5-4 Remote systems view in WDz

Tip: Positioning of views in a perspective is highly customizable in the Eclipse environment. You can drag and drop any view within a perspective. You can also open any additional view you like in a perspective. Go to **Window** → **Show View** → **Other...** to bring up a list of available views.

5.3 Editing MVS file systems with WDz

Using the WDz remote system explorer view, you can edit members in data sets, sequential files, and files under USS. Once a connection is established to the host, you should see MVS Files and USS Files categories under the connection. The contents of the view are controlled using *Filters*. When you connect to a host, you should see some files under each category that are populated using the default filters based on your user name. You can add additional filters to specific files in the host. To add an additional filter to the view:

1. Select either MVS Files or USS Files.
2. From the context menu, select **New** → **Filter**.

3. In the new filter wizard, type the filter to narrow down the selection.
 - a. Type `/usr/lpp/java/J5.0` to view all files in the `/usr/lpp/java/J5.0` folder. You can further narrow down the filter by specifying the subset by file name or subset by file types options.
 - b. Type `SYS1.*` to view all data sets starting with the `SYS1` prefix.

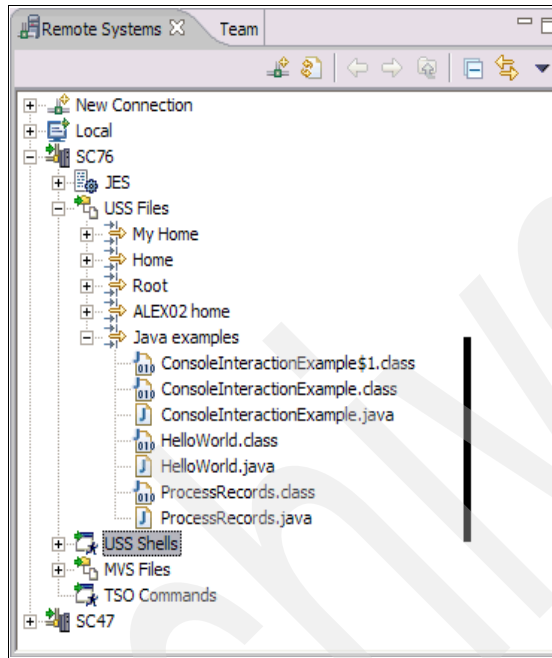


Figure 5-5 Remote system view showing files under USS

Files shown under the filters are editable by double-clicking on a file in the view. WDz starts an appropriate editor depending on the type of file being selected. The mapping between file type and editor is specified in File Association under **Preference** → **Workbench** → **File Association**. In the preference page, you can specify your choice of editor depending on the file extension.

USS files are classified using their extensions. For example, WDz fires up a Java editor when a file with the `.java` extension is opened.

The file type for a member in an MVS data set is determined by the name of the containing data set, which is specified using data set name patterns in the z/OS File System Mapping view. For example, all members in data sets ending with COBOL will have the `.cbl` extension by default. So WDz opens the LPEX editor for such a member. If a data set contains members with different types, it is possible to set the mapping information individually (select a member from the

data set, **Properties**, then select **Mapping** from the context menu to specify the specific extension for the selected member).

5.4 Managing MVS jobs with WDz

The Remote Job Monitor is a z/OS subsystem that provides you with access to your jobs in JES. In the Remote Job Monitor view, you can view job status and output as well as performing cancel, purge, hold, restart and release actions against jobs. Figure 5-6 on page 81 shows a sample of the Remote Job Monitor view showing JES output (right of the image showing output of job JZ0SEXPL). If you double-click on a job, it will display the JES output in a local editor.

The Remote Job Monitor also uses filters to narrow down what's been displayed under a JES heading in the Remote System explorer view. To create a new filter in the view:

1. Select the JES heading in the Remote System view.
2. Select **New** → **New JES Job Filter...** from the context menu.
3. Specify job owner, job prefix, job status, job class, and job output class in the New JES Job Filter wizard.
4. Give a name for the filter.
5. New job information should be displayed in the view.

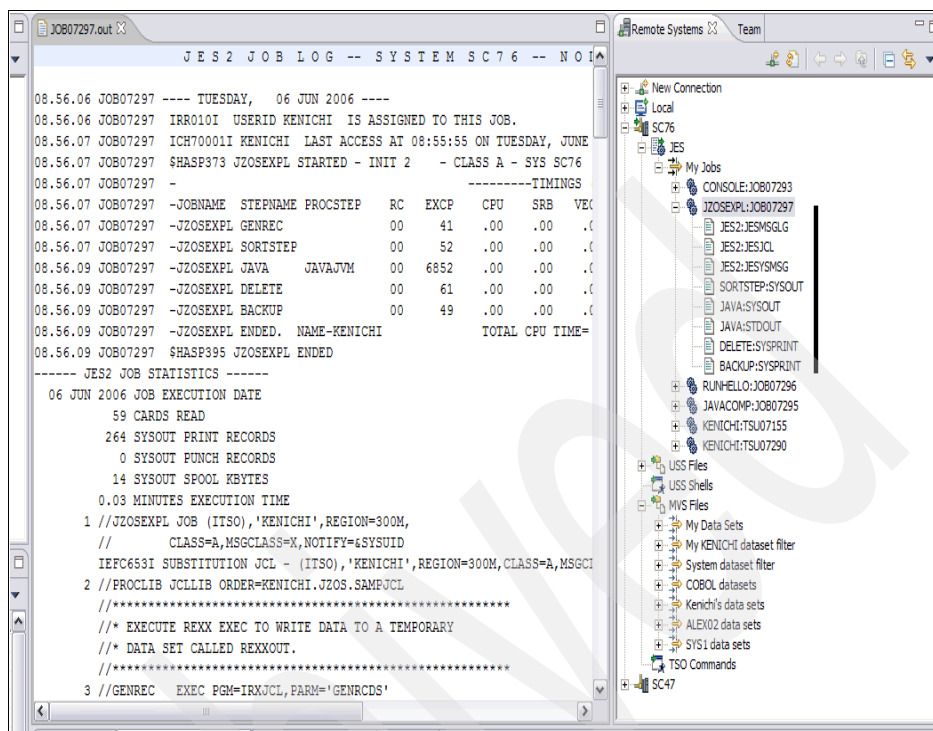


Figure 5-6 Remote Job Monitor and JES output view

5.5 Remote debugging using WDz

The setup required to perform remote debugging of Java applications running on z/OS using Eclipse-based tools is described in detail in Chapter 4 of *Java Stand-alone Applications on z/OS, Volume I*, SG24-7177. The basic steps remain the same in WDz and are summarized here:

1. Develop and transfer the Java application to z/OS in an appropriate format (for example, copy class files, or as .jar file).
2. Write a JCL script that launches the application with the remote debug options:

```
-Xdebug -Xrunjdwp:transport=dt_socket,server=y,address=8000
```
3. Submit the job. By default, the JVM suspends the application until it receives a connection from a client (the *suspend* option controls this behavior).
4. Set break points in the Java application.
5. Configure and start the remote debugger.

6. Step through the program.

When the JZOS launcher is used to start a Java program in a job, the debugging options are set using the IBM_JAVA_OPTIONS environmental variable (one of the environmental variables used in the IBM Java SDK), rather than passing the options as arguments to Java directly. This should be specified as part of STDENV, which is described in Chapter 3, “Job management using JZOS” on page 25. Example 5-1 lists the contents of environmental settings used to launch a Java application in debug mode using JZOS.

Example 5-1 Setting up remote debug options with JZOS

```
. /etc/profile
export JZOS_HOME=/u/kenichi/jzos
export JAVA_HOME=/usr/lpp/java/J5.0

export PATH=/bin:${JAVA_HOME}/bin:

LIBPATH=/lib:/usr/lib:${JAVA_HOME}/bin
LIBPATH="${LIBPATH}:${JAVA_HOME}/bin/classic
LIBPATH="${LIBPATH}:${JZOS_HOME}"
export LIBPATH="${LIBPATH}:"

# Customize your CLASSPATH here
CLASSPATH=/u/kenichi/examples

# Add JZOS required jars to end of CLASSPATH
for i in "${JZOS_HOME}"/*.jar; do
    CLASSPATH="${CLASSPATH}:${i}"
done
export CLASSPATH="${CLASSPATH}:"

# Configure JVM options
IJO="-Xms16m -Xmx128m"
# Debug options.
IJO="${IJO} -Xdebug -Xrunjdw:transport=dt_socket,server=y,address=8000"
export IBM_JAVA_OPTIONS="${IJO} "

export JAVA_DUMP_HEAP=false
export JAVA_PROPAGATE=NO
export IBM_JAVA_ZOS_TDUMP=NO
```

Tip: You can export a .jar file containing your Java project from WDz to USS using the export wizard:

1. Select the Java project to export.
2. Select **Export** from the context menu.
3. Select **Remote Jar file** from the list. Click **Next**.
4. Select the resources to export.
5. Click **Browse** to select a directory on a remote connected system.
6. Press **Finish**.

5.6 Remote debugging of a Java application with JNI calls

We now describe how you can remotely debug a Java application that includes JNI calls to a COBOL program. We use the example described in 6.3.2, “Calling a COBOL program from a Java application” on page 101 to demonstrate how it can be done using the remote debugging facility available in WDz. Steps for doing the remote debugging for such applications are as follows:

1. Create an MVS project in WDz.
2. Include a COBOL program in the project.
3. Include an Assembler macro in the project (Example 5-2 on page 83).
4. Build the entire project using “Rebuild project” from the context menu.
5. Generate a C header file from the Java class that includes the native method.
6. Implement the call to the COBOL program in the C program, and compile under USS.
7. Create the DLL library, which includes COBOL and C programs.
8. Run the Java program in the remote debug mode.

Steps 5 through 8 are described in detail in 6.3.2, “Calling a COBOL program from a Java application” on page 101.

Example 5-2 illustrates an Assembler macro that needs to be assembled together with the COBOL program in order to set up the LE environment for the remote debugging.

Example 5-2 Assembler macro to set up language environment for remote debugging

CEEUOPT CSECT

```
CEEUOPT AMODE ANY  
CEEUOPT RMODE ANY  
CEEXOPT TEST=(,,,"VADTCPIP&&9.65.2.32%8001:*")  
END
```

Java Native Interface (JNI)

In this chapter, we describe the Java Native Interface (JNI) that can be used by Java code to access C or C++ code. And the JNI also works the other way around, enabling C or C++ code to access Java code, as also described here.

We provide the following information:

- ▶ An overview of the Java Native Interface (JNI) in 6.1, “Introducing the Java Native Interface (JNI)” on page 86
- ▶ A simple example showing how to get started in 6.2, “Getting started with a JNI application” on page 93
- ▶ More advanced examples showing how to call an Assembler Callable Service and a COBOL program in 6.3, “More advanced JNI usage” on page 97
- ▶ An explanation of the integration between Java and COBOL in both directions using the support in IBM Enterprise COBOL in 6.4, “Integrating Java and COBOL using IBM Enterprise Object-Oriented COBOL” on page 105

6.1 Introducing the Java Native Interface (JNI)

SDK V5 offers many different APIs to interact with various system resources. However, there are still situations in which an application cannot be written entirely in Java. For such applications, the Java Native Interface (JNI) might provide a solution. Basically, JNI can be used in two ways:

- ▶ It can be used to invoke native functions from your Java application.
- ▶ Or you can call Java methods from your native application.

JNI was originally designed to be used in combination with C/C++, but you can also achieve interlanguage interoperability with Java through COBOL. This function is part of Enterprise COBOL for z/OS V3R4. This method is fairly new and until recently, using JNI was only possible in a combination of Java and C/C++.

So, in most cases JNI is still used in combination with C/C++. Through C/C++ you can access a huge number of z/OS UNIX Assembler Callable Services, which in turn enables you to reach resources that may not be accessible directly through Java code. In that situation you could decide to use a native language to achieve your goal. There is also the possibility of using an indirect interface. By “indirect” we mean, for example, a piece of C/C++ code that provides a link to an Assembler module. That way you can use the JNI to integrate Assembler in your Java application.

Figure 6-1 on page 87 shows an example of a Java program that uses JNI and C to get to an MVS resource. It essentially has three functions:

1. A Java program that is the driver, which writes to the output device. The source code should contain the interaction between Java and the native method invocation. And preferably, it should also contain code to deal with error conditions.
2. A Java class with the native method definition.
3. A DLL module, written in C, that contains the entry points for the basic functions. Those functions are then mapped by the Run Time Library (RTL) to Assembler Callable Services (BPX), which in turn are executed by the z/OS UNIX Kernel. These Callable Services allow UNIX programs to access system resources.

Therefore, the JNI is basically located between the Java Virtual Machine (JVM) and the operating system (z/OS). Instead of going directly from the JVM to the operating system (as the dotted line in Figure 6-1 indicates), we use a native library to access the required resources.

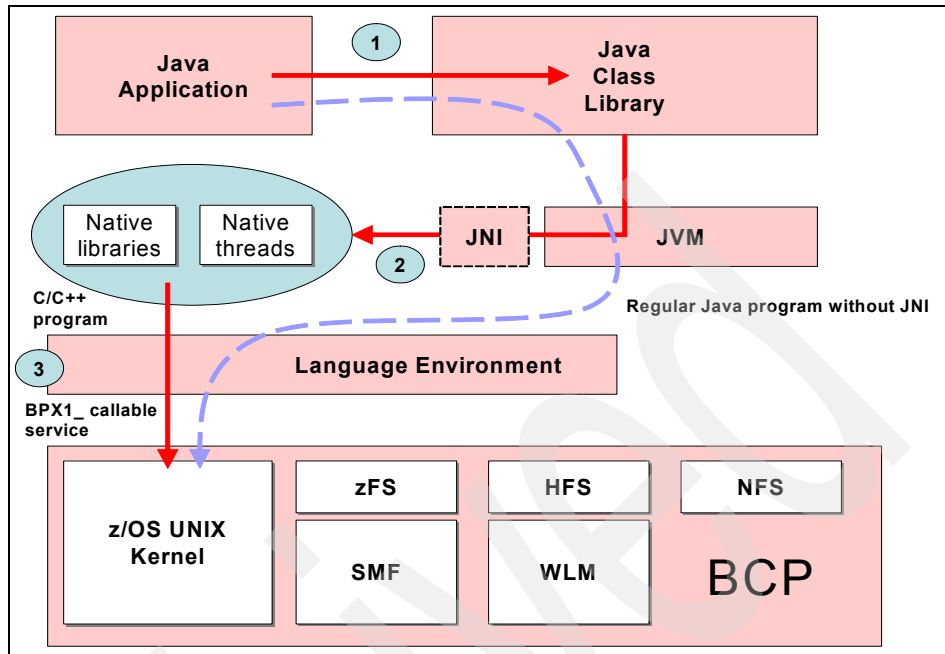


Figure 6-1 Example of using native methods for file I/O

Using JNI offers some advantages over using a 100% pure Java program. For example, it allows you to consider the reuse of code from another language that is already available. In addition, the performance of native code is generally faster than Java code, so some performance gains could be expected.

However, there are also disadvantages to using JNI that you need to consider:

- ▶ Using JNI is not recommended for a J2EE environment such as WebSphere Application Server, because if something goes wrong in your calling program code, it may crash the J2EE server.
- ▶ The z Application Assist Processors (zAAP) can exclusively be used to run Java code. Therefore, in order to benefit from zAAP, it is not recommended to have your application be 99% written in native code and to only use Java to invoke that code via the Java Native Interface (JNI).

Note: Details of programming the JNI is a huge topic, and a complete explanation is beyond the scope of this IBM Redbook. Therefore, in this chapter we only show basic concepts and some z/OS specifics of JNI, concentrating on explaining running examples rather than covering every facet of the JNI.

A basic introduction to Java programming with JNI is available at IBM developerWorks:

<http://www-128.ibm.com/developerworks/edu/j-dw-javajni-i.html>

We discuss our own more advanced samples in 6.2, “Getting started with a JNI application” on page 93.

6.1.1 Basic elements of JNI

In this section we discuss the most important elements in the JNI specification.

JNI primitive type and reference types

There are two data types used in Java to pass arguments and obtain returned values, which enable you to interact between the Java language and Native language. The JNI defines *primitive types* and *reference types* to represent Java language elements.

The primitive types are very straightforward and can be mapped from C/C++ to Java just by adding a j character to the primitive type, as illustrated in Figure 6-2.

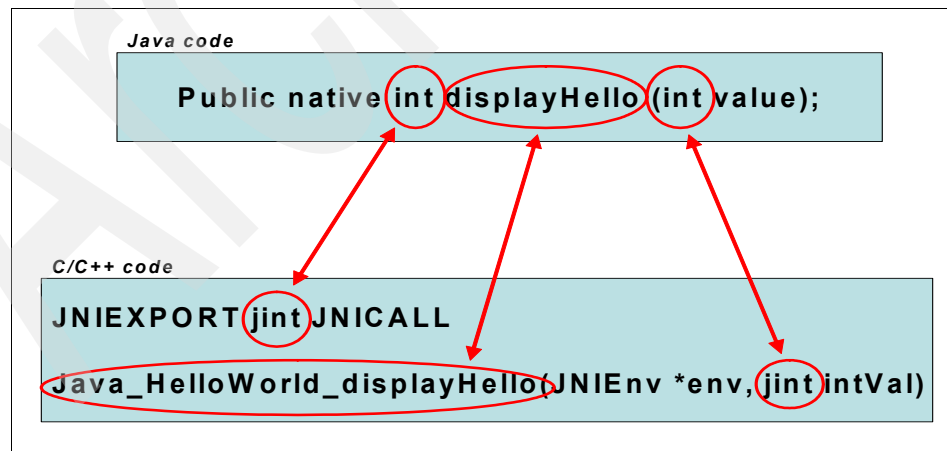


Figure 6-2 Methods mapping

Note: Before you can use the mapping of data types in your C/C++ code, include the `jni.h` header file, as shown in Figure 6-6 on page 95.

Table 6-1 lists the native data types and describes their size and attributes. Use this table when mapping Java native types to C data types in JNI code.

Table 6-1 Mapping of primitive types

| Java type | Primitive type | Description |
|-----------|----------------|------------------|
| Boolean | jboolean | unsigned 8 bits |
| Byte | jbyte | signed 8 bits |
| Char | jchar | unsigned 16 bits |
| Short | jshort | signed 16 bits |
| Int | jint | signed 32 bits |
| Long | jlong | signed 64 bits |
| Float | jfloat | 32 bits |
| Double | jdouble | 64 bits |

JNI reference types correspond to different types of Java objects. Sometimes reference types are also called *object types* (either term is okay to use). They are organized in a hierarchy, as shown in Figure 6-3 on page 90.

In the C programming language, all other objects map to the jobject reference type. Accordingly, a Java String object will be referenced in C as a jstring. A java.util.Hashtable object will be referenced as a jobject.

Object references play an important role when you access Java objects from native code with JNI functions, or when you pass Java objects as parameters to a native implementation.

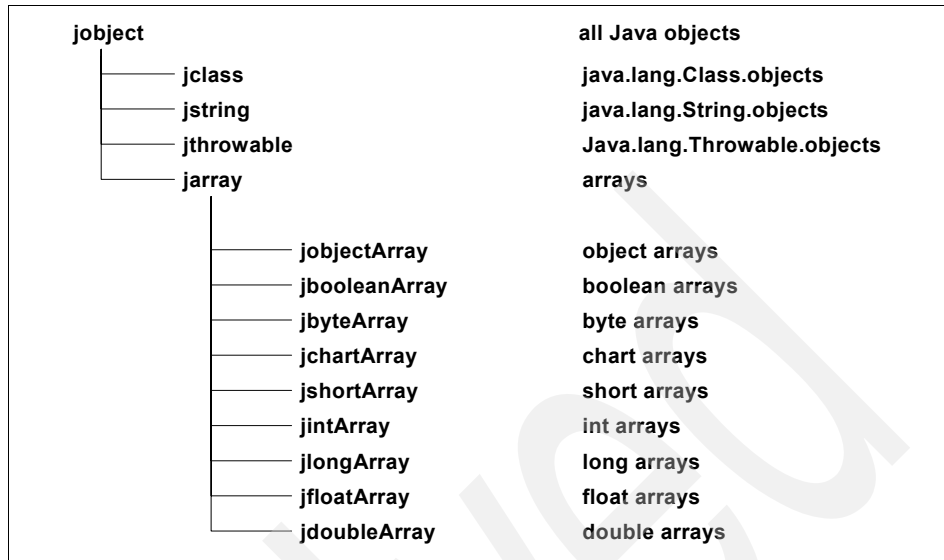


Figure 6-3 JNI reference types

Signature type

Another main data type available in JNI is the signature type. To execute a Java method from native code, you have to specify the exact signature of the method. For example, the Java method:

```
long myMethod (float f, String s, Hashtable[] harr);
```

has the following type signature:

```
(FLjava/lang/String; [Ljava/util/Hashtable;)J
```

Table 6-2 Mapping of signature types to Java types

| Type signature | Java type |
|----------------|-----------|
| Z | boolean |
| B | byte |
| C | char |
| S | short |
| I | int |
| J | long |
| F | float |

| Type signature | Java type |
|--------------------|-----------|
| D | double |
| V | void |
| L <i>classname</i> | class |

This may be considered as a more advanced topic, because calling Java code from within a native program can be very complicated. So if you are not sure about the exact signature of a Java method, you can use this simple method:

1. Create a dummy Java class that defines a native method that has the same arguments and return types. For example:

```
class Helpme {
    native long myMethod (float f, String s, java.util.Hashtable[]
harr);
}
```

2. Compile the Java class.
3. Run the **javah** command on the class:

```
javah -jni Helpme
```

4. The resulting C header file `Helpme.h` contains a description of the native method, including its signature:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Helpme */

#ifdef _Included_Helpme
#define _Included_Helpme
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:      Helpme
 * Method:     myMethod
 * Signature:  (FLjava/lang/String;Ljava/util/Hashtable;)J
 */
JNIEXPORT jlong JNICALL Java_Helpme_myMethod
(JNIEnv *, jobject, jfloat, jstring, jobjectArray);
#ifdef __cplusplus
}
#endif
#endif
```

6.1.2 z/OS 64-bit considerations

In z/OS V1R6, an enhancement was introduced to support 64-bit application development. This included an opportunity to expand the address space up to 1 million TB.

The z/OS 64-bit implementation uses the Long Pointer 64 (LP64) programming model, which is a standard from the Open Group consortium. This introduced more data types that support a length of 8 bytes (64 bits), instead of a 4-byte length data type (32 bits). Refer to the Open Group Web site for further details:

http://www.unix.org/version2/whatsnew/lp64_wp.html

The LP64 model is integrated into the IBM Language Environment (LE) of z/OS. As you already know, the LE provides a single run-time environment for applications written in High Level Languages (HLL), such as COBOL, C/C++, PL/I and Fortran. LE continues support for 32 bit and 24 bit. Support for the 64-bit mode is provided only for C/C++ using the LE run-time environment and for LE conforming assembler, which has no run-time environment.

Using the LP64 model can provide additional benefits. For example, already mentioned is the larger virtual address space size, which allows you expand to 1 million TB. This allows you to put more data in memory, without the need to use files or dataspace, and can be helpful in reducing I/O and its coherent complexity. The LP64 model also provides backward compatibility for the data types. So `int` is still 4 bytes in length in both the 32-bit and 64-bit model.

IBM 64-bit SDK for z/OS

In order to write, test, and deploy Java applications in a 64-bit environment, you need the SDK for z/OS V5, or SDK for z/OS V1.4. The V1.4 level became available in September 2004, so check the maintenance levels to verify that you have the 64-bit version.

When you start out with a 64-bit Java application that contains JNI calls, it is important to verify that the underlying native code is also 64 bit. This is necessary because the native code associated with the 64-bit JVM runs in the same process space as the JVM. It is not possible to mix 64-bit code and 32-bit code, because it that will imply incompatible address space sizes. Therefore, the native code needs to be recompiled and linked in LP64 mode.

Note: Redpaper *z/OS 64-bit C/C++ and Java Programming Environment*, REDP-9110, discusses migrating to 64 bit. It focuses on C/C++ programming in a 64-bit environment, and also deals with Java 64-bit programming.

6.1.3 JNI and the Garbage Collector

The Garbage Collector reclaims garbage. And in the case of JNI, there is a so-called local reference in the stack of the thread that points to an object in the JVM Heap. Figure 6-4 illustrates a thread stack pointing to an object in the JVM. Local references are automatically created and deleted.

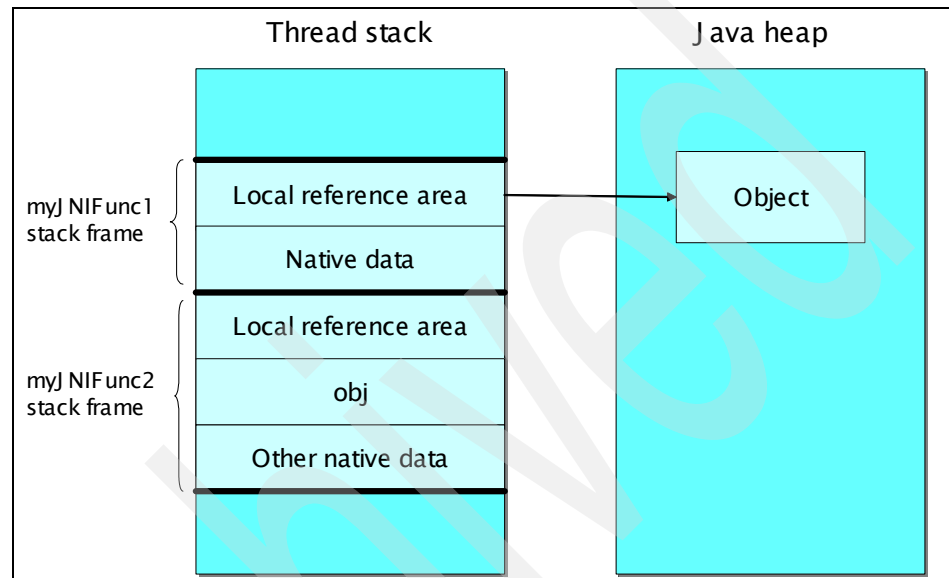


Figure 6-4 Thread stack pointing to an object in the JVM heap

Note: For further information about how Java Native Interface (JNI) works, refer to Chapter 7 of *IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide*, SC34-6650.

6.2 Getting started with a JNI application

You may encounter a situation where you need to use the Java Native Interface (JNI), because no Java functions are available to access the required resource. A native program written in C/C++ can interact with a Java program by using a header file. This header file has a .h extension, and is created by using the **javah** command.

After you obtain the header file, you need to include it in the native source code and create the Dynamic Link Library. In this section we explain the process and provide several samples that you can learn from and build upon.

6.2.1 Developing a simple JNI program

Figure 6-5 and Figure 6-6 on page 95 show two simple sources that make up an JNI program. We will use this section to guide you through the basic steps of developing a JNI program.

Figure 6-5 shows the source of our HelloWorld Java application. It declares a native method and implements the main method.

This native method is the piece that we want to write in C. Therefore, we use the keyword `native` as shown in [1]. The statement `System.loadLibrary`, indicated by [2], loads the shared library (or DLL) containing our native method implementation.

In order for the JVM to find the DLL at run time, the directory where the DLL resides must be part of the `LIBPATH` environment variable in your profile.

```
class HelloWorld {  
    public native void displayHello(); [1]  
    static {  
        System.loadLibrary("HelloJNI"); [2]  
    }  
    public static void main (String[] args) {  
        new HelloWorld().displayHello();  
    }  
}
```

Figure 6-5 Java code for HelloWorld

Figure 6-6 on page 95 shows our C source, which holds the native method implementation. The implementation will be a regular DLL function that is integrated with the Java class.

In this source, [1] is the mandatory header file that defines the JNI interface functions. The line at [2] is the header file that contains the function prototype for the C function that we get after we compile the Java code and run the javah tool. Refer to “Putting it all together” on page 95 for an explanation of the right order in which to set things up.

```
#include <stdio.h>
#include <jni.h>           [1]
#include "HelloWorld.h"   [2]
JNIEXPORT void JNICALL Java_HelloWorld_displayHello(JNIEnv *env,
    jobject obj)
{
    printf("Hello World, this is C, called from Java ...\n");
}
```

Figure 6-6 C code for HelloWorld

Putting it all together

To put everything together we can use the script in Figure 6-7, which can be invoked by issuing the following command:

```
make -f <makefile>
```

First, we compile the Java source in a straightforward way. After that we create a C-Header file with the **javah -jni** command. The javah tool comes with SDK V5. It creates a C/C++ header file from the HelloWorld class, providing a function prototype for the implementation of the native method `displayHello()`.

In our example, javah generates a file HelloWorld.h containing a function prototype `Java_HelloWorld_displayHello`. After you obtain a header file, use the C compiler to compile the .c file and the .h file into a shared library (dll).

```

MAIN = HelloWorld
CFLAGS = -W "c,langlvl(extended)" -W "c,float(ieee)"
IFLAGS = -I. -I/usr/lpp/java/J5.0/include
LL = -W c,dll -W c,exportall
SIDEDECKPATH = /usr/lpp/java/J5.0/bin/j9vm
SIDEDECK = libjvm
$(MAIN).class: $(MAIN).java
.javac $(MAIN).java
$(MAIN).h: $(MAIN).class
.javah -jni -o $*.h $(MAIN)
$(MAIN): $(MAIN).o
.c++ -o libHelloJNI.so $(MAIN).o $(SIDEDECKPATH)/$(SIDEDECK).x
$(MAIN).o: $(MAIN).c $(MAIN).h
.c++ -c -o $(MAIN).o $(LL) $(CFLAGS) $(IFLAGS) $(MAIN).c

```

Figure 6-7 Makefile for HelloWorld

If everything is set up correctly, our JNI example application will display:

Hello World, this is C, called from Java ...

Summary of basic steps to set up this JNI program:

1. Write the Java code.
2. Compile the Java source with the **javac** command.
3. Create a C-header file with the **javah** command.
4. Write the native method implementation in a C source file.
5. Create a Shared Library (DLL) by compiling the .h file and the .c file that was created before.
6. Run the program with the **java** command.

Troubleshooting

Keep the following points in mind when developing this program:

- ▶ Ensure that the program has been compiled as a dll and that the execute permission bit is set. Otherwise, you may receive EDC5207S errors saying Load request for DLL load module unsuccessful.
- ▶ Ensure that `System.loadLibrary()` matches your shared library name, and that it can be found via `LIBPATH`.

6.3 More advanced JNI usage

In the following sections we demonstrate more advanced usage with JNI.

6.3.1 Calling system services from a Java application

The z/OS operating system provides a large set of low-level system services. These services are typically made available as either assembler macros or C library functions. You can refer to the following documentation for more detailed information about these services:

- ▶ *MVS Programming: Assembler Services Guide*, SA22-7605-06
- ▶ *UNIX System Services Programming: Assembler Callable Services Reference*, SA22-7803-08
- ▶ *z/OS V1R7.0 MVS Programming: Authorized Assembler Services Guide*, SA22-7608-09

These services typically require special authorization and are designed to work under special conditions. For example, *z/OS C/C++ Programming Guide*, SC09-4765-06, describes constraints about z/OS UNIX System Services:

Access to z/OS UNIX System Services is intended to be through the z/OS UNIX System Services XL C/C++ run-time library only. The z/OS XL C/C++ compiler does not support the direct use of z/OS UNIX System Services callable services such as the assembler interfaces. You should not directly use z/OS UNIX System Services Callable services from your z/OS XL C/C++ application programs, because problems can occur with the processing of the following:

- *Signals*
- *Library transfers*
- *fork()*
- *exec()*
- *Threads*

There are comparable z/OS XL C/C++ run-time library functions for most z/OS UNIX System Services callable services, and you should use those instead. Do not call assembler programs that access z/OS UNIX System Services callable services.

So you can see that it is important to understand all the special requirements for services you might use in your applications. In the following section, we describe how you can call such services from Java applications using JNI.

Description of scenario (accessing SMF records)

The System Management Facility (SMF) collects and records system- and job-related information that your installation can use in:

- ▶ Billing users
- ▶ Reporting reliability
- ▶ Analyzing the configuration
- ▶ Scheduling jobs
- ▶ Summarizing direct access volume activity
- ▶ Evaluating data set activity
- ▶ Profiling system resource use
- ▶ Maintaining system security

In the following section, we demonstrate how to interface with SMF from a Java application using JNI. Specifically, we demonstrate how to use one of the services in the z/OS UNIX System Services to identify which SMF types are currently monitored by the system.

Note that SMF types of 0 through 127 are reserved for IBM products. So, in this scenario, we describe a Java program that reports which IBM products are currently monitored. For further information about SMF and SMF types, refer to *MVS System Management Facilities (SMF)*, SA22-7630-12.

Development steps

Here we describe the step-by-step method for developing a Java application that interacts with SMF using system services provided by z/OS. Although we describe the steps required to interact with SMF, this process should be similar for using any other services, as well.

Special considerations for smf_record (BPX1SMF, BPX4SMF)

- ▶ The `smf_record` callable service writes an SMF record to the SMF data set.
- ▶ The caller must be permitted to the BPX.SMF FACILITY class profile (that is, the owner of a job calling this service must have the authority to run the program successfully).
- ▶ The service can also be used to determine if a particular type or subtype of SMF record is being recorded (this is the functionality we need to determine which SMF types are currently monitored on the system).
- ▶ Use the C/C++ runtime library rather than calling the service directly.

Description of SMF service call

The *z/OS C/C++ Run-Time Library Reference*, SA22-7821-07, provides a description of a C wrapper function to call SMF service. The signature of the function is as follows:


```
int __smf_record(int smf_record_type,
                int smf_record_subtype,
                int smf_record_length,
                char *smf_record);
```

The `__smf_record()` function writes an SMF record pointed to by `smf_record` of length `smf_record_length` for SMF record type `smf_record_type` and subtype `smf_record_subtype` to the SMF data set.

SMF record types of 0 through 127 are reserved by IBM products, and the layout of each SMF record is documented in *MVS System Management Facilities (SMF)*, SA22-7630-12. If 0 (NULL) is passed as an argument for `smf_record`, the function determines if the specified SMF record type is being monitored in the current system configuration.

If the call to the function is successful, it returns 0. If unsuccessful, it returns -1.

The function is declared in a header file called *unistd.h*.

Declaring a native function in a Java class

In the Java class definition, a declaration of a native function must be included in order to call such functions. Example 6-1 shows a declaration of a Java class which includes a native method (`checkSmfRecord()`).

Example 6-1 Example of native function declaration

```
public class ServiceCallExample {
    // Declaration of native method.
    public native void checkSmfRecord();

    // Loading a library containing the native method.
    static {
        System.loadLibrary("MyLib");
    }

    public static void main(String args[]) {
        System.out.println("Starting...");
        ServiceCallExample e = new ServiceCallExample();
        // Calling the native method
        e.checkSmfRecord();
        System.out.println("Done!");
    }
}
```

Generating a header file for native methods

A C header file containing the function prototype of the native method needs to be generated based on the declaration of such function in the Java source file. After compiling the Java source, generate the header file using the following command:

```
javah YourJavaClassName
```

For the declaration of this native method, it generates the following signature for the native method:

```
JNIEXPORT void JNICALL
    Java_ServiceCallExample_checkSmfRecord(JNIEnv *, jobject);
```

Implementing a native method

Based on the signature generated, you can implement the functionality in the C programming language utilizing the system service, as shown in Example 6-2.

Example 6-2 C function which makes service call to interact with SMF

```
#include <stdio.h>
#include <jni.h>
#include <unistd.h>
#include "ServiceCallExample.h"

JNIEXPORT void JNICALL
    Java_ServiceCallExample_checkSmfRecord(JNIEnv *e, jobject jo) {
    int rc, i;

    for (i = 0; i < 128; i++) {
        rc = __smf_record(i, 0, 0, NULL);
        if (rc == -1) {
            printf("Type %d: Not monitor or error\n", i);
        } else {
            printf("Type %d: Monitored\n", i);
        }
    }
}
```

Compiling the native method

You can compile the native method and create a library as follows:

```
c89 -c -o ServiceCallExample.o -W "c,langlvl(extended)" -W
c,expo,dll -I/usr/lpp/java/J5.0/include ServiceCallExample.c
```

```
c89 -W l,dll -o libMyLib.so ServiceCallExample.o
```

Running the example

You need to make sure that the library is accessible when running the Java application. For example, if the library and Java class exists in the same directory in USS, then you need to include the current directory (that is, ".") in the LIBPATH environmental variable:

```
export LIBPATH=${LIBPATH}:.  
export CLASSPATH=${CLASSPATH}:.  

```

After these environmental variables are set correctly, you should be able to run the application.

6.3.2 Calling a COBOL program from a Java application

In this section, we demonstrate how to call a COBOL program from a Java application. The Java program can be invoked via the command prompt under USS, or it can be submitted as a batch job.

Overview

Because Java Native Interface (JNI) only enables you to call C/C++ programs, a *glue layer*, which facilitates the invocation of a COBOL program, is also required; see Figure 6-8 on page 102. A C glue layer program is able to call COBOL programs using the z/OS Language environment (LE) Inter-Language Communication (ILC) mechanism, which is described in detail in "z/OS Language Environment Writing Interlanguage Communication Applications", SA22-7563.

LE enables easy interlanguage communication among supported high-level languages including C, C++, COBOL, and PL/1. Using the method described here, you should be able to call any existing programs regardless of their implementation language.

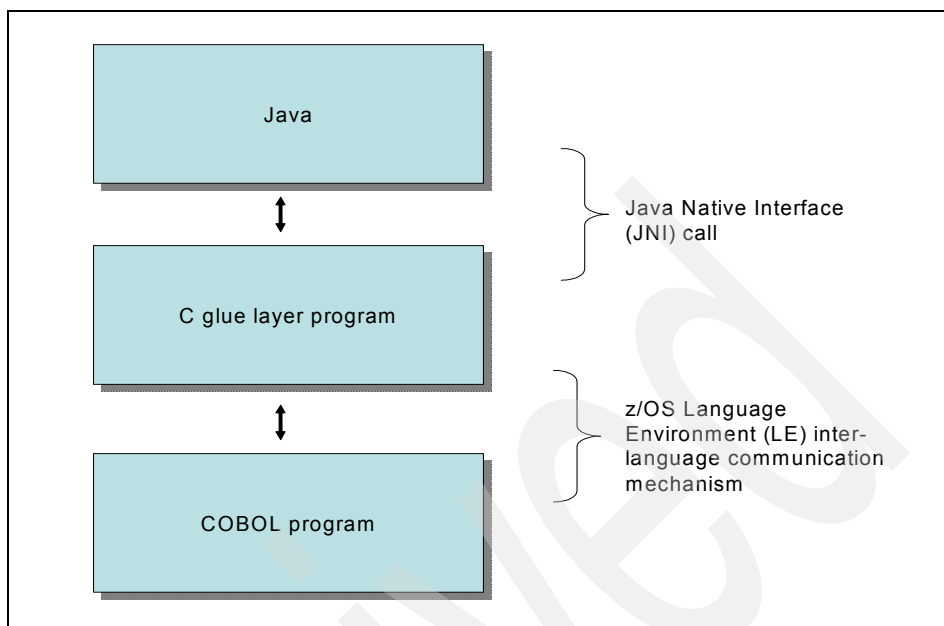


Figure 6-8 Overview of Java program calling COBOL program

Development steps

Next we present an overview of the steps required to call a COBOL program from a Java application. For the purposes of illustration, we show how to call a COBOL implementation of the HelloWorld! program from a Java application.

Developing the COBOL application

Example 6-3 on page 102 illustrates a simple COBOL program which prints a “Hello World from COBOL program!!” message.

Example 6-3 COBOL program called by Java application

```

IDENTIFICATION DIVISION.
PROGRAM-ID. MYCOB1
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
    DISPLAY "Hello World from COBOL program!!"
    GOBACK.
  
```

Example 6-4 on page 103 illustrates a JCL script, which compiles and link-edit the COBOL program.

Note that the source of the COBOL program exists in a standard MVS data set ('KENICHI.SOURCE.COBOL(MYCOB1)'), and the load module for this program is created in another standard MVS data set ('KENICHI.PGM.LOAD.OBJ(MYCOB1)'). When we create a dynamic link library, we will point to this data set.

Example 6-4 JCL to compile and link-edit COBOL program

```
//KENICHIA JOB (ITS0),'kenichi',CLASS=A,MSGCLASS=X,MSGLEVEL=(1,1),
// REGION=4096K,TIME=(5,0),NOTIFY=&SYSUID
//COMPILE EXEC PGM=IGYCRCTL,REGION=2048K,
// PARM=RENT
//STEPLIB DD DSN=IGY.SIGYCOMP,DISP=SHR
//SYSIN DD DSN=KENICHI.SOURCE.COBOL(MYCOB1),DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSN=&&LOADSET,UNIT=SYSDA,
// DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
// DCB=(BLKSIZE=3200)
//SYSUT1 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT6 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//*****
//LKED EXEC PGM=HEWL,COND=(8,LT,COMPILE),REGION=1024K,
// PARM='LIST,XREF,RMODE(ANY),RENT'
//SYSLIB DD DSN=CEE.SCEELKED,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSN=&&LOADSET,DISP=(OLD,DELETE)
// DD DDNAME=SYSIN
//SYSLMOD DD DSN=KENICHI.PGM.LOAD.OBJ(MYCOB1),DISP=SHR
//SYSUT1 DD UNIT=SYSDA,SPACE=(TRK,(10,10))
```

Tip: The data set containing the load module of the COBOL program must have the appropriate suffix (such as .OBJ) in order to create a DLL library later. Otherwise, you will see an error message such as the following:

```
FSUM3010 Specify a file with the correct suffix (.C, .hh, .i, .c,
.i, .s, .o, .x, .p, .I, or .a), or a corresponding data set name,
instead of //'KENICHI.LOAD(MYCOB1)'.
```

Developing the Java application

The Java application only needs to know the name of C function that invokes the COBOL program. Such function can be declared as a native method which is described in detail in the previous section. Example 6-5 shows a sample Java program that includes the declaration of a native method (that is, a function in C glue program).

Example 6-5 Java program calling C glue program to invoke COBOL program

```
public class Java2C2CobolExample {
    // C native method which calls COBOL program.
    public native void callCobol();
    static {
        System.loadLibrary("MyLib");
    }

    public static void main(String args[]) {
        System.out.println("Hello World from Java program!");
        Java2C2CobolExample e = new Java2C2CobolExample();
        e.callCobol();
    }
}
```

Based on the native method declaration, you can generate the prototype for the function in the C glue program using the **javah** utility program.

Developing the C glue layer

In the C glue layer, you need to define the function based on the signature generated by the **javah** utility program so that it can be called from the Java application. In addition, you need to declare the name of the COBOL program as an external function using an **extern** declaration.

With such declaration of the COBOL program, it is quite simple to invoke the COBOL program from the C glue layer. Figure 6-6 shows a sample C glue program that invokes a COBOL program.

Example 6-6 Example of the C glue program to call a COBOL program

```
#include <stdio.h>
#include <jni.h>
#include "Java2C2CobolExample.h"

/* Declaration of COBOL program. */
extern void MYCOB1(void);

/* Implementation of JNI C function. */
```

```
JNIEXPORT void JNICALL
Java_Java2C2CobolExample_callCobol(JNIEnv *e, jobject o) {
    printf("Hello World from C program!!\n");
    MYCOB1();
}
```

Creating a Dynamic Link Library

In this step, you need to create a Dynamic Link Library (DLL) which contains the C glue program and the COBOL program described in earlier section. You can create a DLL file by using the following commands:

```
c89 -c -o Java2C2CobolExample.o -W "c,langlvl(extended)"
-W c,expo,dll -DNEEDSIEEE754 -DNEEDSLONGLONG
-I/usr/lpp/java/J5.0/include Java2C2CobolExample.c

c89 -W l,dll -o libMyLib.so Java2C2CobolExample.o
"// 'KENICHI.OB(MYCOB1)'"
```

The first command compiles and creates an object file for the C glue layer. The second command creates a DLL library file (libMyLib.so) that contains both C and COBOL programs.

Running the example

After completing these steps, you can invoke the Java program from a command prompt under USS or submit it as a batch job. Either way, you need to place the DLL library somewhere visible so that it can be found during execution.

The easiest approach is to set the LIBPATH environmental variable to include the directory where you generate the DLL file. If everything is set up correctly, you should see output like the following:

```
Hello World from Java program!
Hello World from C program!!
Hello World from COBOL program!!
```

6.4 Integrating Java and COBOL using IBM Enterprise Object-Oriented COBOL

The example in 6.3.2, “Calling a COBOL program from a Java application” on page 101 shows an example using a glue layer in C. This approach is quite complex and error-prone, but it is required if you want to call a traditional (non-OO) COBOL program from Java.

In the following sections we describe how Java and COBOL can be integrated in an easier way by using the new object-oriented syntax in IBM Enterprise COBOL for z/OS and enhancements in the COBOL compiler.

Important: The sample programs used throughout this chapter are included in the additional material for this IBM Redbook. See Appendix A, “Additional material” on page 147 for details.

6.4.1 IBM Enterprise COBOL for z/OS introduction

IBM Enterprise COBOL for z/OS supports the new object-oriented syntax, which makes the integration of Java applications and COBOL programs much simpler. The syntax is based upon the facilities of the Java Native Interface, the primary means that Java provides for interoperating with non-Java programs. Object-oriented COBOL enables you to define Java classes, with methods implemented in COBOL.

For a detailed discussion of object-oriented COBOL programming, refer to *Enterprise COBOL for z/OS Programming Guide*, SC27-1412, which is available from:

<http://www-306.ibm.com/software/awdtools/cobol/zos/library/>

In the remainder of this section, we provide simple examples of object-oriented COBOL programs which demonstrate interoperability of Java applications and COBOL programs.

6.4.2 Preparing object-oriented applications under UNIX

Here we describe how integrated Java/COBOL application should be prepared in the UNIX environment.

Compiling object-oriented applications under UNIX

When you compile object-oriented applications in a UNIX shell, use the **cob2** command to compile COBOL client programs and class definitions, and the **javac** command to compile Java class definitions to produce bytecode (suffix .class).

To compile COBOL source code that contains object-oriented syntax such as INVOKE statements or class definitions, or that uses Java services, you must use these compiler options: RENT, DLL, THREAD, and DBCS. (The RENT and DBCS options are defaults.)

A COBOL source file that contains a class definition must not contain any other class or program definitions.

When you compile a COBOL class definition, two output files are generated:

- ▶ The object file (.o) for the class definition.
- ▶ A Java source program (.java) that contains a class definition that corresponds to the COBOL class definition. Do *not* edit this generated Java class definition in any way. If you change the COBOL class definition, you must regenerate both the object file and the Java class definition by recompiling the updated COBOL class definition.

If a COBOL client program or class definition includes the JNI.cpy file by using a COPY statement, specify the include subdirectory of the COBOL install directory (typically /usr/lpp/cobol/include) in the search order for copybooks. You can do this by using the -I option of the **cob2** command, or by setting the SYSLIB environment variable.

Preparing object-oriented applications under UNIX

Use the **cob2** command to link object-oriented COBOL applications.

To prepare an object-oriented COBOL client program for execution, link the object file with the following DLL side files to create an executable module:

- ▶ libjvm.x, which is provided with your IBM Java 2 Software Development Kit.
- ▶ igzcjava.x, which is provided in the lib subdirectory of the COBOL directory in the HFS. The typical complete path is /usr/lpp/cobol/lib/igzcjava.x. This DLL side file is also available as member IGZCJAVA in the SCEELIB PDS (part of Language Environment).

To prepare a COBOL class definition for execution, follow these steps

1. Link the object file using the DLL side files to create an executable DLL module.

You must name the resulting DLL module to libClassname.so, where Classname is the external class-name. If the class is part of a package and thus there are periods (.) in the external class-name, you must change the periods to underscores (_) in the DLL module name.

For example, if class Account is part of the com.acme package, the external class name (as defined in the REPOSITORY paragraph entry for the class) must be com.acme.Account, and the DLL module for the class must be libcom_acme_Account.so.

2. Compile the generated Java source with the Java compiler to create a class file (.class).

For a COBOL source file `Classname.cbl` that contains the class definition for `Classname`, you would use the following commands to compile and link the components of the application:

Table 6-3 Commands for compiling and linking a class definition

| Command | Input | Output |
|--|-----------------------------|---|
| <code>cob2 -c -qdll,thread Classname.cbl</code> | <code>Classname.cbl</code> | <code>Classname.o</code> <code>Classname.java</code> |
| <code>cob2 -bdll -o libClassname.so Classname.o libjvm.x igzcjava.x</code> | <code>Classname.o</code> | <code>libClassname.so</code> |
| <code>javac Classname.java</code> | <code>Classname.java</code> | <code>Classname.class</code> |

After you issue the **cob2** and **javac** commands successfully, you have the executable components for the program:

- The executable DLL module `libClassname.so`
- The class file `Classname.class`

All files from these commands are generated in the current working directory.

Tip: It is recommended that you compile, link and run applications that use object-oriented syntax in the z/OS UNIX environment. However, it is also possible to compile, prepare and run OO applications by using standard batch JCL or TSO/E commands.

6.4.3 Java program calling COBOL

The following example demonstrates how to call a COBOL program from a Java application. The enterprise COBOL compiler generates a wrapper Java class that calls the actual COBOL programs via JNI. That is, when you compile the `Hello.cbl` source file with the enterprise COBOL compiler, it generates a `Hello.java` class, which on its turn can be used in your own Java calling application.

As far as the Java application is concerned (`Driver.java`, in this example), the COBOL program can be called as if it is calling a method in a class (via `Hello.java` class).

Example 6-7 on page 109 shows the Java application invoking the Java “wrapper” class for the COBOL program.

Example 6-7 Driver.java: example of Java program invoking COBOL program

```
class Driver {
    public static void main(String[] args) {
        System.out.println("Just about to call COBOL...");
        Hello H = new Hello();
        H.sayHello();
        System.out.println("Done...");
    }
}
```

Example 6-8 shows the COBOL program being called.

Example 6-8 Hello.cbl: Enterprise COBOL program which can be called easily from Java application

```
cbl dll,thread,rent
Identification Division.
Class-id. Hello inherits Base.
Environment Division.
Configuration section.
Repository.
    Class Base is "java.lang.Object"
    Class Hello is "Hello".
Identification Division.
Object.
Data division.
Working-storage section.
Procedure division.
    Identification Division.
    Method-id. "sayHello".
    Data division.
    Linkage section.
    Procedure division.
        Display "Hello World, from 00 COBOL!".
    End method "sayHello".
End object.
End class Hello.
```

6.4.4 COBOL program calling Java

Example 6-9 on page 110 illustrates how to call a method defined in a Java class from an object-oriented COBOL program. When you are running this example, the Language Environment (LE) runtime option XPLINK(on) must be set. You can invoke the program with this option as follows:

```
_CEE_RUNOPTS="XPLIN(ON)" ./Driver
```

Alternatively, this option can be set globally using **export** as shown here:

```
export _CEE_RUNOPTS="POSIX(ON)"
```

Example 6-9 Driver.cbl

```
cbl dll,thread,rent,lib,pgmname(longmixed)
Identification division.
Program-id. "Driver" recursive.
Environment division.
Configuration section.
Repository.
    Class Hello is "Hello".
Data Division.
Working-storage section.
01 anObj usage object reference Hello.
Procedure division.
    Display "Just about to call Java method..."

    Invoke Hello New returning anObj
        On exception
            Display "Error creating Java object..."
        Stop run
    End-invoke

    Display "Invoke Java instance method"
    Invoke anObj "sayHello"
        On exception
            Display "Error invoking java"
        Not on exception
            Display "Java method returned"
    End-invoke

    Display "Done..."
    Goback.
End program "Driver".
```

Example 6-10 on page 110 shows the output.

Example 6-10 Hello.java

```
public class Hello
    extends java.lang.Object {
    public void sayHello(){
        System.out.println("Hello from Java");
    }
}
```

I/O using the JZOS toolkit API

In this chapter, we discuss the functions of the JZOS toolkit API. We also compare best practices for using the JZOS toolkit for I/O compared to the Java Record I/O (JRIO) APIs, and show you examples of the new options available to developers using the JZOS toolkit.

Note: The JRIO library is covered in Chapter 3 of *Java Stand-alone Applications on z/OS, Volume I*, SG24-7177. We highly recommend reading that chapter before proceeding in this chapter.

7.1 JZOS toolkit library introduction

The JZOS toolkit library is a complementary set of functions to the Java Record I/O library(JRIO). Together these libraries allow Java applications access to mainframe file systems that ordinary Java APIs do not support.

The JZOS toolkit provides thin wrappers for the z/OS C/C++ Library functions, which can be used to access MVS data sets. For a full discussion on these C/C++ functions, refer to the following IBM C/C++ publications:

- ▶ *z/OS C/C++ Run-Time Library Reference*, SA22-7821-07
- ▶ *z/OS C/C++ Programming Guide*, SC09-4765-06

Using the JZOS toolkit, Java programs can access any MVS data sets supported by the C/C++ library, including:

- ▶ Partitioned Data Set (PDS)
- ▶ Partitioned Data Set Extended (PDSE)
- ▶ Sequential Files
- ▶ Virtual Sequential Access File (VSAM) of the type KSDS, RRDS, or ESDS

The z/OS C/C++ library supports several models of I/O when using MVS data sets:

- | | |
|--------------------|--|
| Record mode | Each read or write processes a single record of a data set. |
| Stream mode | Data set records are presented as a stream of bytes. Each read or write reads some portion of those bytes, irrespective of record boundaries. Stream mode is further distinguished by two types: Text (stream) mode Data set records are converted to a stream of bytes and a “new line” record delimiter is placed in the stream between records after trailing blanks are removed. Binary (stream) mode Data set records are placed in the stream as is. |

In the following sections we present an overview of the functions that make up the JZOS toolkit library, including examples on how to access both PDS and VSAM data sets using text stream and record mode.

7.2 JZOS classes

In this section, we discuss the JZOS classes `ZFile`, `ZUtil`, `FileFactory`, `PdsDirectory` and `PdsDirectory.Member`. The JZOS classes associated with WTO are discussed in 3.4.2, “JZOS APIs for console communication” on page 44.

The `ZFile` class contains all the JNI wrapped C/C++ functions for working with data sets. `ZFile` useful static methods include the following.

| | |
|---|--|
| <code>allocDummyDDName()</code> | Allocates and returns a new <code>DDName</code> , allocated to “DUMMY”. |
| <code>bpxwdyn(command)</code> | Calls the <code>BPXWDYN</code> service, which is a text-based interface to dynamic allocation. |
| <code>ddExists(ddname)</code> | Answer true if the given DD name exists. |
| <code>dsExists(dsn)</code> | Answer true if the given data set name exists. |
| <code>getDefaultHLQ()</code> | Returns the default data set name high-level qualifier for the user/job. |
| <code>getFullyQualifiedDSN(jdsn)</code> | Returns a fully qualified data set name. |
| <code>getSlashSlashQuotedDSN(dsn)</code> | Given a data set name, answers the fully-qualified data set name enclosed in single quotes and preceded by a double forward slash mark <code>//</code> . |
| <code>remove(fileName)</code> | Removes (deletes) a file (a data set). |
| <code>rename(oldName, newName)</code> | Renames a file (a data set). |

Tip: Data sets should be accessed with a preceding double forward slash mark `//`. If a member name or HLQ is referred to when accessing the data set, include single quotes as shown here:

```
//'USER1.DATA(MEMBER)'
```

This data set name syntax is exactly what is used in the C/C++ library.

`ZFile.getSlashSlashQuotedDSN(dsn)` will append the HLQ to the data set and then prepend double slashes and insert single quotes.

`ZFile.getFullyQualifiedDSN(dsn)` will remove single quotes and the double slashes. If the argument does not include single quotes, the HLQ will be prepended to it.

7.3 Usage examples

In the following sections we provide a number of samples using the JZOS toolkit APIs.

- ▶ In 7.3.1, “Copying members of a data set using JZOS FileFactory” on page 114 and 7.3.2, “Copying members of a data set using ZFile” on page 117, we show how you can copy members of an MVS data set.
- ▶ In 7.3.3, “BPXWDYN” on page 118, we show an example of how to call a callable service called BPXWDYN.
- ▶ In “Virtual Sequential Access Method (VSAM) overview” on page 120, we explain how to work with VSAM files.

7.3.1 Copying members of a data set using JZOS FileFactory

The example program in Example 7-1 on page 115 uses the JZOS toolkit API to copy all members of one data set into a new data set. It uses the `PdsDirectory` and `PdsDirectory.Member` classes to create a listing of all PDS members and static methods from the `ZFile` class to manipulate the data set representation strings. `BufferedReader` and `BufferedWriter` wrap around the JZOS `FileFactory` class to read data from the members into the new members. This is an example of text stream processing.

Invoke the program as follows:

```
java MemberCopy USER.DSN
```

The first argument to the program specifies the data set to be copied. Members are then copied from the source data set into `USER.DSN.COPY`. When the program is successful, it prints out the new members as they are copied and the number of lines copied. When the program completes, it prints out the total members copied.

The program checks to make sure the source data set exists using the static method `ZFile.exists(//USER.DSN1)`. It then uses `PdsDirectory` and `PdsDirectory.MemberInfo.getName` to generate an `ArrayList` of string names of all the members in a data set, and a corresponding member in a copy.

That array list is passed to the method `copyMembers`. The method `copyMembers` steps through the array list by twos, wrapping a `FileFactory` around a `BufferedReader` and `BufferedWriter` to read from the source data set member and create a copy of it.

Example 7-1 Member copy using JZOS FileFactory

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;

import com.ibm.jzos.FileFactory;
import com.ibm.jzos.PdsDirectory;
import com.ibm.jzos.ZFile;

// Copies all members of PDS to another data set.
public class MemberCopy {

    public static void main(String args[]) {

        if (args.length != 1) {
            System.err.println("Usage: MemberCopy SourceDSN");
            System.exit(8);
        }
        try {
            copyMembers(memberList(args[0]));
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(8);
        }
    }

    public static ArrayList memberList(String inputDsn) {
        ArrayList members = new ArrayList();

        try {

            //split inputDsn to make it
            //easy to remove the HLQ
            //since ZFile.getSlashSalshQuotedDSN adds it
            String spInputDsn[] = inputDsn.split("\\.");
            inputDsn = null;
            inputDsn = spInputDsn[1];

            for (int i = 2; i < spInputDsn.length; i++) {
```

```

        inputDsn += "." + spInputDsn[i];
    }

    String outputDsn = inputDsn + ".COPY";
    if (!ZFile.exists(ZFile.getSlashSlashQuotedDSN(inputDsn)))
    {
        System.out.println("input dsn does not exist!");
        System.exit(8);
    }

    if (ZFile.exists(ZFile.getSlashSlashQuotedDSN(outputDsn)))
    {
        System.out.println("output dsn already exists!");
        System.exit(8);
    }

    PdsDirectory dir = new PdsDirectory("//" + inputDsn);
    int i = 0;
    for (Iterator iter = dir.iterator(); iter.hasNext();) {
        PdsDirectory.MemberInfo info =
(PdsDirectory.MemberInfo) iter.next();
        String fullInputDsn = inputDsn + "(" + info.getName() +
        + ")";
        String fullOutputDsn = outputDsn + "(" + info.getName()
        + ")";
        members.add(fullInputDsn);
        members.add(fullOutputDsn);
    }
    } catch (IOException io) {
        io.printStackTrace();
    }
    return members;
}

public static void copyMembers(ArrayList members) {

    BufferedReader rdr = null;
    BufferedWriter wtr = null;
    int count = 0;

    try {
        for (int i = 0; i < members.size(); i += 2) {
            String input = ZFile.getSlashSlashQuotedDSN((String)
members.get(i));

```

```

        String output = ZFile.getSlashSlashQuotedDSN((String)
members.get(i + 1));
        //Jzos file factory
        rdr = FileFactory.newBufferedReader(input);
        wtr = FileFactory.newBufferedWriter(output);

        String line;
        long llines = 0;
        while ((line = rdr.readLine()) != null) {
            wtr.write(line);
            wtr.write("\n");
            llines++;
        }
        count++;
        System.out.println("Copied " + input + " lines=" +
llines);

        if (wtr != null)
            wtr.close();
        if (rdr != null)
            rdr.close();
    }
} catch (IOException io) {
    io.printStackTrace();
}

System.out.println(count + " members copied to PDS");

}

}

```

7.3.2 Copying members of a data set using ZFile

Using a FileFactory to work with data sets falls into the Java programming paradigm. Now we are going to present the same example as presented in Example 7-1 on page 115, but in this case we use ZFile functions.

Example 7-2 on page 118 shows a replacement for the method copyMembers. Here, as we step through the array list of member names, we are using the static method ZFile.getSlashSlashQuotedDSN to create a fully qualified DSN string.

We then create a buffer the exact size of the record length using the method getLrec1. Finally, we use read() and write() methods to copy bytes from the source data set to the destination data set.

Example 7-2 Member copy using JZOS ZFile

```
public static void copyMembers(ArrayList members) {

    try {
        int count = 0;
        for (int i = 0; i < members.size(); i += 2) {
            String input = ZFile.getSlashSlashQuotedDSN((String)
members.get(i));
            String output = ZFile.getSlashSlashQuotedDSN((String)
members.get(i + 1));
            ZFile zFileIn = new ZFile(input,
"rb,type=record,noseek");
            ZFile zFileOut = new ZFile(output,
"wb,type=record,noseek");
            //Jzos file factory

            byte[] recBuf = new byte[zFileIn.getLrecl()];
            int nRead;
            while ((nRead = zFileIn.read(recBuf)) >= 0) {
                zFileOut.write(recBuf, 0, nRead);
            }

            System.out.println("Copied " + input + " LRECL=" +
zFileOut.getLrecl());
            zFileIn.close();
            zFileOut.close();
            count++;
        }
        System.out.println(count + " members copied to PDS");
    } catch (IOException io) {
        io.printStackTrace();
    }
}
```

7.3.3 BPXWDYN

The callable service BPXWDYN is an interface to a subset of dynamic allocation and dynamic output services. BPXWDYN supports data set allocation,

unallocation, concatenation, the retrieval of certain allocation information, and the addition and deletion of output descriptors. BPXWDYN can be called from REXX, C, Assembler, PL/I and now—through the JZOS toolkit—it can be called from Java. For more information on BPXWDYN, refer to *z/OS V1R7.0 Using REXX and z/OS UNIX System Services*, SA-7806-08. Example 7-3 is a Java program that will allocate a new data set with all the attributes of an existing data set using the callable service BPXWDYN. Invoke the program as follows:

```
java DynallocDataset "//USER1.SOURCE.DSN" "//USER1.NEW.DSN"
```

From the example, you can see that to invoke BPXWDYN just call the static method `bpxwdyn` from the `ZFile` class a standard BPXWDYN input string. This example allocates a new data set using the “like” keyword.

Example 7-3 Allocating a data set using BPXWDYN

```
import com.ibm.jzos.ZFile;

public class DynallocDataset {

    public static void main(String[] args) throws Exception {

        if (args.length < 2) {
            System.out.println("Usage: existing Dataset newDataset");
            System.exit(8);
        }

        // Allocate new DDNAME to "DUMMY" that we will reuse to allocate
        the new dataset
        String targetDD = ZFile.allocDummyDDName();
        // bpxwdyn requires fully-qualified dsn (it will not add uid).
        // so we make sure that we have a fully qualified DSNs
        String sourceDSN = ZFile.getFullyQualifiedDSN(args[0]);
        String targetDSN = ZFile.getFullyQualifiedDSN(args[1]);

        // Allocate the output dataset using BPXWDYN.

        ZFile.bpxwdyn("alloc fi(" + targetDD + ") da(" + targetDSN
            + ") like(" + sourceDSN + ") reuse new catalog msg(wtp)");
        try {
            ZFile.bpxwdyn("free fi(" + targetDD + ") msg(wtp)");
        } catch (Exception ignore) {}
    }
}
```

7.3.4 Working with VSAM files

In the following sections we explain how to access VSAM files using the JZOS toolkit APIs.

Virtual Sequential Access Method (VSAM) overview

Virtual Sequential Access Method (VSAM) data sets were designed by IBM to give users the ability to store data in sequential, random, and keyed access data set types. The different types of VSAM data sets are:

- ▶ Key-Sequenced Data Set (KSDS)
- ▶ Entry-Sequenced Data Set (ESDS)
- ▶ Relative Record Data Set (RRDS)
- ▶ Linear Data Set (LDS)

The JZOS toolkit library can only access KSDS, ESDS and RRDS VSAM data sets. VSAM data sets are organized into *logical records*. A record may contain fields of data associated with a client, inventory, and so on.

Of equal importance to the information stored in a logical record is the *key*. The key is used to retrieve a specific logical record.

A KSDS *cluster* includes a data component and an index component. Clusters simplify access to VSAM data sets by allowing the data component and index component to be referred to with their own, single cataloged name.

Example 7-4 on page 121 is a sample JCL that demonstrates how to use IDCAMS to create a VSAM cluster file. This cluster file can be used in conjunction with Example 7-5 on page 122 to illustrate how to use JZOS to read, write, update and delete records from a VSAM file.

This cluster file is set up with a 80-byte record length, and an 8-byte key field. Running the sample JCL with a rc(0) will yield the following:

| | |
|------------------------------|--------------|
| USER1.PRIVATE.CLUSTER | Cluster File |
| USER1.PRIVATE.KSDS.IX | IndexFile |
| USER1.PRIVATE.KSDS.DA | Data File |

Java applications refer to VSAM files by their cluster names. There is no facility included in the JZOS library to allocate a VSAM data set. To work with VSAM data sets in JZOS, the data sets must be allocated before working with them.

```
//VSAM JOB (ITS0),'JBARNEY',CLASS=A,
// NOTIFY=&SYSUID,MSGCLASS=T
/* -----
/* Create VSAM file.
/* -----
//VSAMALLOC EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
    DEFINE CLUSTER (NAME('USER1.PRIVATE.CLUSTER') -
        TRK(4 4) -
        VOL(xxxxx) -
        RECSZ(80 80) -
        INDEXED -
        NOREUSE -
        KEYS(8 0) -
        OWNER(USER1) ) -
    DATA -
        (NAME(USER1.KSDS.DA)) -
    INDEX -
        (NAME(USER1.KSDS.IX))

/*
```

Working with VSAM records using the JZOS library

The JZOS library provides several JNI wrapped C++ function calls to manipulate VSAM data sets. Refer to the C++ Programmer's guide for more information about processing VSAM files with the C library.

The `ZFile` class contains many methods for adding, deleting and updating records in a VSAM data set. It also includes informational methods to find out such things as logical record length (LRECL). The obvious difference between `ZFile` access to a VSAM data set and `JRIO IKeyedAccessReference` is `byteStream` access.

Using `ZFile`, you can read a portion of a record, which can be useful for reading large records. The following example shows how to read a record from a VSAM file.

```
ZFile zfile = new ZFile(filename, options);
zfile.locate(key, offset, keyLen, ZFile.LOCATE_KEY_EQ));
int nRead = zfile.read(recBuf);
```

The record is located in the data set by its key and the offset into the start of the key buffer, and then read into a byte array `recBuf`. All access to records for reading, writing, updating and deleting have to start with reading the record.

In the example, the `read` method is returning the number of bytes read. If `read` is called again, it will return the next record and the number of bytes read, or it will return `-1`, if EOF is encountered.

Commonly used `ZFile` methods for working with VSAM data sets are:

| | |
|-------------------------|---|
| getLrecl | Get the native file's logical record length. |
| getVsamKeyLength | Get the <code>__vsamkeylen</code> from the C-library <code>fldata</code> structure. |
| getVsamType | Return the VSAM type. |
| locate | Locate a record given a key. |
| locate | Locate a record using its record number or RBA. |
| read | Read from the native file into the supplied buffer. |
| update | Update a VSAM record. |
| write | Write the buffer to the native file. |
| delrec | Delete the current VSAM record. |

Like the equivalent JRIO methods `read(..)`, `write(..)` and `update(..)`, the JZOS methods take as input a byte array of the same length as the record to be worked with. Unlike the JRIO methods for working with VSAM records, however, the JZOS methods can also take as input a byte array of any size.

Example 7-4 on page 121 shows how to set up a VSAM cluster which the sample program shown here in Example 7-5 uses.

Example 7-5 Java program to manipulate a VSAM KSDS file

```
import java.io.UnsupportedEncodingException;
import com.ibm.jzos.ZFile;
import com.ibm.jzos.ZUtil;

public class ZFileKSDS {
    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.err.println("Usage: ZFileKSDS //'SourceDSN'");
            System.exit(8);
        }

        if (!ZFile.exists(args[0])) {
            System.out.println("input dsn does not exist!");
            System.exit(8);
        }
    }
}
```



```

String filename = args[0];
String options = "ab+,type=record";
ZFile zfile = new ZFile(filename, options);
int lrec1 = zfile.getLrec1();
int keyLen = 8;

try {
    // construct some records with key prefixes
    byte[] rec_1 = padToLength("AAAAAAAARecord 1",
lrec1).getBytes(ZFile.DEFAULT_EBCDIC_CODE_PAGE);
    byte[] rec_2 = padToLength("BBBBBBBBRecord 2",
lrec1).getBytes(ZFile.DEFAULT_EBCDIC_CODE_PAGE);
    byte[] rec_3 = padToLength("CCCCCCCCRecord 3",
lrec1).getBytes(ZFile.DEFAULT_EBCDIC_CODE_PAGE);
    byte[] recBuf = new byte[lrec1];
    byte[] partialRecBuff = new byte[10];
    int nRead, nUpdated;
    // write some records
    zfile.write(rec_1);
    zfile.write(rec_2, 0, rec_2.length); // alternate form
    zfile.write(rec_3);

    // point to the first record
    zfile.locate(rec_1, 0, keyLen, ZFile.LOCATE_KEY_EQ);

    // read back the record and verify its contents
    nRead = zfile.read(recBuf);
    System.out.println("Record1 \t\t:" +
printBytes(recBuf,nRead));

    // update the record
    byte[] rec_1U = padToLength("AAAAAAAARecord 1 updated",
lrec1).getBytes(
        ZFile.DEFAULT_EBCDIC_CODE_PAGE);
    nUpdated = zfile.update(rec_1U);

    // read back the record and verify its contents

    zfile.locate(rec_1, 0, keyLen, ZFile.LOCATE_KEY_EQ);
    nRead = zfile.read(recBuf);
    System.out.println("Record1 After Update \t:" +
printBytes(recBuf,nRead));

```

```

        // point to the second record, using alternate form of
locate()
        byte[] keybuf = new byte[keyLen];
        System.arraycopy(rec_2, 0, keybuf, 0, keyLen);

        zfile.locate(keybuf, ZFile.LOCATE_KEY_EQ);

        // read back the record and verify its contents
        nRead = zfile.read(recBuf);
        System.out.println("Record2 \t\t:" +
printBytes(recBuf,nRead));

        // update the second record, using the alternate update()
form
        byte[] rec_2U = padToLength("BBBBBBBBRecord 2 updated",
lrec1).getBytes(
            ZFile.DEFAULT_EBCDIC_CODE_PAGE);
        nUpdated = zfile.update(rec_2U, 0, rec_2U.length);

        // read back the record and verify its contents

        zfile.locate(rec_2, 0, keyLen, ZFile.LOCATE_KEY_EQ);
        nRead = zfile.read(recBuf);
        System.out.println("Record2 After Update \t:" +
printBytes(recBuf,nRead));

        //point to the third record
        zfile.locate(rec_3, 0, keyLen, ZFile.LOCATE_KEY_EQ);

        // read back a portion of the record
        nRead = zfile.read(partialRecBuff,0,8);
        System.out.println("Partial Record"+nRead+ "\t:" +
printBytes(partialRecBuff,partialRecBuff.length));

        // update the record
        byte[] partialUpdate =
padToLength("CCCCCCCCUpdatedSometingExtra",lrec1).getBytes(
            ZFile.DEFAULT_EBCDIC_CODE_PAGE);
        nUpdated = zfile.update(partialUpdate,0,80);

        //point to the third record
        zfile.locate(rec_3, 0, keyLen, ZFile.LOCATE_KEY_EQ);

        // read back the record and verify its contents
        nRead = zfile.read(recBuf);

```

```

        System.out.println("Record3 Update \t:" +
        printBytes(recBuf,nRead));

        // delete all of the records
        zfile.locate(rec_1, 0, keyLen, ZFile.LOCATE_KEY_EQ);
        nRead = zfile.read(recBuf); // have to read a rec b4
updating
        zfile.delrec();

        zfile.locate(rec_2, 0, keyLen, ZFile.LOCATE_KEY_EQ);
        nRead = zfile.read(recBuf);
        zfile.delrec();

        zfile.locate(rec_3, 0, keyLen, ZFile.LOCATE_KEY_EQ);
        nRead = zfile.read(recBuf);
        zfile.delrec();

    } finally {
        zfile.close();
    }
}

/**
 * Pad a string with spaces to a specified length
 */
static String padToLength(String s, int len) {
    StringBuffer sb = new StringBuffer(len);
    sb.append(s);
    for (int i = s.length(); i < len; i++)
        sb.append(' ');
    return sb.toString();
}

/**
 * Check a condition and throw an Exception with message if not
true
 */
static void check(String msg, boolean value) {
    if (!value)
        throw new RuntimeException(msg);
}

/**
 * Print out bytes as text

```

```

    */
    private static StringBuffer printBytes(byte[] bytes, int nRead) {
        String encoding = null;
        StringBuffer sb = null;

        try {
            encoding = ZUtil.getDefaultPlatformEncoding();
            sb = new StringBuffer(new String(bytes, 0, nRead,
encoding));
        } catch (UnsupportedEncodingException uee) {
            uee.printStackTrace();
        }

        return sb;
    }
}

```

7.4 JZOS versus JRIO

The JZOS toolkit API and the JRIO library both contain methods for performing I/O in the z/OS environment. They are complementary to each other and provide you not only with a choice of how to access and work with files and data sets, but also with a choice about the style of access.

The JRIO library is modeled to be complementary to the standard Java I/O packages, thereby giving you as a Java developer the ability to work with HFS files and data sets in a Java-like paradigm.

Alternatively, the JZOS toolkit API contains JNI wrapped C/C++ methods that allow programmers who are more familiar with C/C++ I/O on z/OS to use that paradigm for their I/O needs.

The list provided in Table 7-1 on page 127 indicates when to use JZOS.

Table 7-1 When to use JZOS

| Type of access required | API to use |
|---|------------------------|
| C/C++ library interface | JZOS |
| Java data set record stream abstractions | JRIO |
| Fine access to system error codes | JZOS |
| Data set access (Text Stream, Binary Stream, and Record mode) | JZOS |
| Data set access (Record mode) | JRIO |
| Portable text file processing (HFS) | JZOS |
| Portable text file processing (data sets) | JZOS |
| VSAM data set access (KSDS, ESDS, RRDS) | JZOS, JRIO (KSDS only) |
| HFS access | JRIO, java.io |

Java problem determination

In this chapter, we present various problem scenarios and describe how to debug such problems. For more information and guidelines, refer to *Java Stand-alone Applications on z/OS, Volume I*, SG24-7177; it is available at:

<http://www.redbooks.ibm.com/redbooks/pdfs/sg247177.pdf>

We also discuss the changes in Java 1.5 as compared to Java 1.4.x with respect to problem determination. Finally, we provide details about the Diagnostic Tooling Framework for Java (DTFJ).

8.1 Typical Java problems

Most of the problems faced by Java developers, either during application development or during production runs, can be broadly categorized into the following types:

- ▶ OutOfMemory problems
- ▶ Hang or deadlock problems
- ▶ Crash problems
- ▶ High CPU or performance problems

In the following sections, we explain these challenges in more detail.

8.1.1 OutOfMemory problems

Java applications running in a JVM can have two kinds of OutOfMemory issues:

- ▶ Exhaustion of LE HEAP.

This can be caused by a leak in the usage of LE HEAP. JVM uses LE HEAP for two types of allocations:

- For allocation of Java heap. During startup, the JVM allocates a single chunk of maximum Java heap from LE HEAP.
- The JVM uses LE HEAP for its native memory allocations, where the JVM typically uses *malloc* to allocate the memory. It is also possible for any JNI code that is running under a Java process to allocate memory from LE HEAP.

In order to debug any memory leaks with respect to usage of LE HEAP by JVM or JNI code, users can use JVM options such as `-Xrunjchck` and `-memorycheck`. For example:

```
java -memorycheck application
```

- ▶ When Java heap exhausts and there is no space for object allocation, the JVM throws an OutOfMemory error. The best way to monitor Java heap usage is by enabling the **verbosegc** option:

```
java -verbosegc application
```

The following reasons can result in Java heap exhaustion.

- When the application requirement of Java heap itself is large when compared to the current maximum heap setting (Xmx), it can result in an OutOfMemory situation. In these cases the problem can be overcome by increasing Xmx. For example:

```
java -Xmx512m application
```


- The other reason that can result into an OutOfMemory situation is Java heap fragmentation. One way to avoid this is to start with a small minimum heap (Xms). It is better to set Xms far lower than Xmx, or to leave Xms at the default value (although this may not help with all fragmentation issues).
- Memory leaks also result in Java heap exhaustion. Generally this is an application bug, but it is also possible that JVM could be causing a memory leak. The best way to start the analysis of these issues is to analyze the heapdump generated. There are various tools available to analyze the heapdumps; refer to the following site:

<http://www-1.ibm.com/support/docview.wss?uid=swg24009436>

A detailed description of how to use the Heap Analyzer tool can be found in the IBM Redbook *Java Stand-alone Applications on z/OS, Volume 1*, SG24-7177, which is available at the following site:

<http://www.redbooks.ibm.com/redbooks/pdfs/sg247177.pdf>

8.1.2 Hang or deadlock problems

A hung process is generally unresponsive, and it can occur because of the following reasons:

- The process has reached a deadlock situation and there is no movement of threads. A deadlocked process does not use processor time, which can be determined using the USS **ps** command against a Java Process ID (PID), for example:

```
ps -p 50332135
```

| PID | TTY | TIME | CMD |
|----------|----------|------|-----------------------------|
| 50332135 | ttyp0000 | 8:09 | /usr/lpp/java/J5.0/bin/java |

If the value of TIME increases as the process is executing, then the process is not deadlocked.

A user can also take Java dump to find the cause of the deadlock. A Java dump can be generated using the following command:

```
kill -3 PID
```

In order to analyze a Java dump to diagnose a deadlock issue, refer to the section “Locks, monitors, and deadlocks (LOCKS)” in the chapter titled “Using Javacore” in *IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide*, SC34-6650:

<http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnosis/diag50.pdf>

- If the process is not in a deadlock but there is no response from the process and it is consuming CPU, then the process could be looping. To debug this kind of problem, take a console dump. For more details refer to the section titled “The process is looping” in *IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide*, SC34-6650:

<http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnosis/diag50.pdf>

8.1.3 Crash problems

A crash can occur only because of a JVM internal fault or because of JNI (native) code of the application running under the Java process. When a crash occurs, the JVM produces by default a transaction dump (system dump), CEEDUMP, JVM trace snap dump, and javacore.

When a crash occurs, the following output will be written to stderr:

```
Type=Segmentation error vmState=0x00000000
J9Generic_Signal_Number=00000004 Signal_Number=0000000b
Error_Value=00000000 Signal_Code=00000000
Handler1=19679868 Handler2=19CF22C0
gpr0=00000044 gpr1=19730990 gpr2=19130D80 gpr3=00000001
gpr4=211F55E0 gpr5=00000001 gpr6=19130D80 gpr7=99C5AF2C
gpr8=011AC090 gpr9=00000000 gpr10=00000001 gpr11=06FA31A0
gpr12=191391E0 gpr13=19730910 gpr14=86FA34C0 gpr15=007FF530
psw0=078D1400 psw1=86FA34C0
fpr0=41C229D6 fpr1=3FFF1999 fpr2=3F796F31 fpr3=99999999
fpr4=3F733333 fpr5=3FF00000 fpr6=3D4CCCD0 fpr7=00000000
fpr8=00000000 fpr9=00000000 fpr10=00000000 fpr11=00000000
fpr12=00000000 fpr13=00000000 fpr14=00000000 fpr15=00000000
Program_Unit_Name=
Program_Unit_Address=00000000
Target=2_30_20060606_06679_bHdSMr (z/OS 01.07.00)
CPU=s390 (4 logical CPUs) (0x100000000 RAM)
JVMDUMP006I Processing Dump Event "gpf", detail "" - Please Wait.
JVMDUMP007I JVM Requesting System Dump using
'KAILAIK.JVM.TDUMP.KAILAIK3.D060614.T110726'
IEATDUMP in progress - Please Wait.
IEATDUMP success for
DSN='KAILAIK.JVM.TDUMP.KAILAIK3.D060614.T110726'
CEEDUMP in progress - Please Wait.
CEEDUMP success for
FILE='/u/kailaik/CEEDUMP.20060614.110741.83886832'
```

```

JVMDUMP010I System Dump written to
KAILAIK.JVM.TDUMP.KAILAIK3.D060614.T110726
JVMDUMP007I JVM Requesting Snap Dump using
'/u/kailaik/Snap0001.20060614.110726.
83886832.trc'
JVMDUMP010I Snap Dump written to
/u/kailaik/Snap0001.20060614.110726.83886832.tr
c
JVMDUMP007I JVM Requesting Java Dump using
'/u/kailaik/javacore.20060614.110726.
83886832.txt'
JVMDUMP012E Error in Java Dump:
/u/kailaik/javacore.20060614.110726.83886832.txt
JVMDUMP013I Processed Dump Event "gpf", detail "".

```

This output gives information about the signal that caused the dump, information about general purpose and floating point registers, and the location of the various dump files. Users can configure the dumps to be generated and the location of the dumps by using facilities provided by JVM. For more information about this topic, refer to Chapter 22 and 23 of *IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide*, SC34-6650:

<http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnosis/diag50.pdf>

In order to determine the failing function, CEEDUMP and javacore can be used. These provide information about the failing thread stack trace. Based on the stack trace, the user can determine whether the failure is inside the JVM or in the native (JNI) code.

If the user suspects the problem to be in the JIT compiler, the user can disable the JIT compiler using the option `-Xint`. If the problem is solved by disabling the JIT, then the user can do further problem determination using various functionalities provided by JIT. Refer to Chapter 29 of *IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide*, SC34-6650:

<http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnosis/diag50.pdf>

The transaction dump also can be used to obtain stack information. Tools that can be used to analyze the transaction dump are:

- The Interactive Problem Control System (IPCS)

The Interactive Problem Control System (IPCS) is a tool in z/OS that helps to diagnose failures.

For more details on the usage of IPCS, refer to *IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide*, SC34-6650:

<http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnosis/diag50.pdf>

► **Dump Formatter**

The Dump Formatter is a cross-platform tool that can be used to analyze dumps on a different platform (for example, a dump generated on AIX can be analyzed on Windows).

The Dump Formatter has two components:

– **jextract**

When a transaction dump is generated, invoke the **jextract** utility. Here is the usage:

```
Usage: jextract [dump_name] [output filename] [[options]
output filename defaults to {dump_name}.zip
or {dump_name}.xml if -nozip option specified
options:
    -help          print this screen
    -nozip         don't create a zip
```

Following is an example:

```
jextract KAILAIK.JVM.TDUMP.KAILAIK3.D060614.T110726
```

jextract generates an xml file and adds the dump and the xml file to a zip file. **jextract** must be run on the same system where the dump is generated. The user can use the second component of the Dump Formatter to analyze the zip file.

– **jdumpview**

Unzip the zip file generated using **jextract** into a separate directory to make sure the dump is not overwritten. You can invoke the **jdumpview** utility on the dump, as follows:

```
jdumpview KAILAIK.JVM.TDUMP.KAILAIK3.D060614.T110726
```

Performance of **jdumpview** can be improved by having enough memory available for paging. You can increase the memory using **Xmx** as follows:

```
jdumpview -J-Xmx KAILAIK.JVM.TDUMP.KAILAIK3.D060614.T110726
```

Because **jdumpview** internally invokes a JVM, the user can use the **-J** option to pass command line arguments to the JVM.

jdumpview provides various commands to analyze memory and work with classes, objects, heap dumps and traces.

For more details about Dump Formatter, refer to Chapter 28 of *IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide*, SC34-6650:

<http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnostics/diag50.pdf>

8.1.4 High CPU or performance problems

Various factors affect CPU usage or performance of an application. Users can use the Resource Management Facility (RMF™) of z/OS to monitor CPU, memory, and I/O device performance.

JVM heap sizing is one of the most important factors that can affect performance. If the Java heap is not set properly, it can lead to excessive GC, fragmentation, and paging, which affects the performance of the application. The output of **verbosegc** can help you to understand the heap usage and Garbage Collector tuning. Refer to Chapter 2 of *IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide*, SC34-6650, for more information about this topic:

<http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnostics/diag50.pdf>

The chapter will help you to understand the Garbage Collector, how to interpret **verbosegc**, as well as how to tune the Java heap to obtain maximum performance from JVM heap usage.

Just In Time (JIT) compilation also plays a vital role in application performance. JIT is switched on by default, but users can disable it by using the option **-Xint**. To improve the performance of short-running applications, users can use **-Xquickstart**.

8.2 Diagnostic Tooling Framework for Java (DTFJ)

Diagnostic Tooling Framework for Java (DTFJ) is an IBM Java Application Programming Interface (API). Using this API, a user can write Java diagnostic tools. Because DTFJ is written in Java, tools developed using DTFJ can be cross-platform. Tools written using DTFJ API can access the following information:

1. Heap locations in the dump.
2. Java threads and native threads in the dump.
3. Details of the machine, java version being used.

4. Java classes and objects in the heap.

For more details about DTFJ, refer to Chapter 37 of *IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide*, SC34-6650:

<http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnosis/diag50.pdf>

8.3 Guidelines - using Java and tuning options

In the following sections we provide guidelines for using Java and we explain tuning options.

8.3.1 Guidelines for using 31-bit versus 64-bit Java

The guidelines to consider when using 31-bit or 64-bit Java are as follows:

1. If an application requires a Java heap larger than the maximum provided by 31-bit Java, or if the application must communicate with a 64-bit process/database, then the application is a candidate for using the 64-bit Java.
2. Increasing the addressability from 31 bits to 64 bits also results in larger Java objects. Therefore, the same Java application will require a larger Java heap which may result in longer Garbage Collection pause times.

8.3.2 Java5 tuning options

Tuning the JVM to obtain optimum performance has been a challenging task. Here we provide details about Java5 functionalities to improve the performance of different kinds of applications:

- Garbage collection (GC) policy

Java5 provides different garbage collection mechanisms, each of which has advantages and drawbacks, depending on the kind of load the application generates.

Generally the default GC policy is sufficient for most applications. However, if application performance is problematic (for example, if longer GC pause times or heap fragmentation is experienced), then users can use the following Java5 GC mechanisms to obtain better performance from GC.

- gencon

This GC policy can be set using `Xgcpolicy` as follows:

```
java -Xgcpolicy:gencon application
```

The **gencon** option results in combined usage of *concurrent* and *generational* GC to reduce the pause times in garbage collection. This GC policy is useful for applications that create short-lived objects (for example, a transactional application).

- optthruput

This is the default policy. This is useful for most applications; it provides high throughput to the application, but at the cost of occasional pauses.

- optavgpause

This GC policy can be set as follows:

```
java -Xgcpolicy:optavgpause application
```

This trades high throughput for short pauses. With this policy, both concurrent mark and concurrent sweep gets enabled. This policy is helpful to applications running with large heaps.

- subpool

This GC policy can be set as follows:

```
java -Xgcpolicy:subpool application
```

This option uses an improved object allocation algorithm to achieve better performance when allocating objects on the heap. This option might improve performance on large SMP systems.

- Large Object Area (LOA)

As Garbage Collector allocates and frees objects, the Java heap can become fragmented in such a way that the heap required for object allocations can only be met by time-consuming compactions. This kind of situation is most likely to happen if the application allocates large objects (> 64 KB).

To reduce fragmentation caused by large objects, the JVM provides a facility to set LOA. JVM provides **-Xloainitial** and **-Xloamaximum** command line options to set initial and maximum LOA.

When allocating the object, the JVM tries to allocate in the Small Object Area (SOA). If it is not able to find a suitable chunk and if the allocation request is equal to or greater than 64 KB, then it tries to allocate in LOA. If there is insufficient space in LOA, or if the allocation request is less than 64 KB, then allocation failure is triggered.

For more details about Garbage Collector, refer to *IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide*, SC34-6650:

<http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnosis/diag50.pdf>

► Class sharing

The new Shared Classes feature in the IBM implementation of version 5.0 of the Java platform offers a completely transparent and dynamic means of sharing all loaded classes. This Shared Classes feature is supported on all platforms on which the IBM implementation of the JVM ships. This feature stores all system and application classes in a persistent dynamic class cache in shared memory.

You enable class sharing by adding the following line to an existing Java command line:

```
-Xshareclasses[:name=<cachename>]
```

When the JVM starts up, it looks for a class cache of the name given (if no name is provided, it chooses a default name). It then either connects to an existing cache or creates a new one, as required.

You specify the cache size using the following parameter:

```
-Xscmx<size>[k|m|g]
```

This parameter only applies if a new cache is created by the JVM. If this option is omitted, a platform-dependent default value is chosen (typically, 16 MB). For more details on class sharing, refer to:

<http://www-128.ibm.com/developerworks/java/library/j-ibmjava4>

► Just In Time compiler (JIT)

The Just In Time compiler provides various functionalities to tune the performance of the JVM:

– -Xcodecache<size>

This option deals with performance. It sets the size of each block of memory that is allocated to store native code of compiled Java methods. By default, this size is selected internally according to the CPU architecture and the capability of your system.

– -Xjit : optlevel=[noOpt | cold | warm | hot | veryHot | scorching]

This option forces the JIT to compile all methods at a specific optimization level.

– -Xquicstart

This option is used for improving startup time of the applications. It is appropriate for short-running applications.

For more detailed information about JIT compiler tuning, refer to *IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide*, SC34-6650:

<http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnosis/diag50.pdf>

Java Virtual Machine Tool Interface

In this chapter, we provide details about the Java Virtual Machine Tool Interface (JVMTI), and give a simple example to demonstrate the use of JVMTI. We also give tips about migrating from the Java Virtual Machine Profiler Interface (JVMPI) to JVMTI.

9.1 Java Virtual Machine Tool Interface (JVMTI)

The Java Virtual Machine Tool Interface (JVMTI) is a two-way native programming interface. It replaces the Java Virtual Machine Debug Interface (JVMDI) and JVMPI deprecated interfaces, although JVMPI-based tools continue to work with 5.0 SDK.

JVMTI provides various facilities to users for writing tools to profile, debug, and monitor various activities of the application through JVM. The JVMTI agent runs in the same process executing the virtual machine, and directly communicates with the virtual machine. Developers are strongly encouraged to use JVMTI instead of JVMDI and JVMPI.

JVMTI agents can be loaded using one of the following options:

- ▶ Use the agent path to specify the agent:
-agentpath:<path-to-agent>=<options>
- ▶ Use the agent library to specify the agent:
-agentlib=<agent>=<options>

Then add the agent path to the LIBPATH.

9.2 Simple example to demonstrate usage of JVMTI

This section explains how to write an agent to debug and profile applications. The agent uses the API provided by the interface to communicate with the application. It can register for getting notification on the events to query, monitor, and control the application. The agent can be written in the C language.

Here are the steps to write a simple example:

1. JVMTI function, event or data type is defined in jvmti.h. So you must include the SDK directory containing jvmti.h in the include path. Then add the following line:

```
#include<jvmti.h>
```

2. After the JVM has loaded the agent, it looks for the startup routine Agent_OnLoad in the library. Therefore, any agent you write must include this start-up routine:

```
JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *vm, char *options,  
void *reserved)
```

The JVM will call the startup routine with <options> passed as the second parameter.

3. The agent may export shutdown function Agent_OnUnload:

```
JNIEXPORT void JNICALL Agent_OnUnload(JavaVM *vm)
```

This API can be used to delete resources when the agent is unloaded. This function will be invoked before the JVM exits.

4. In the agent, you must enable several capabilities to use JVMTI functions and events. For example, to use exception-related functions and events, the parameter can_generate_exception_events must be true.

```
static jvmtiEnv *jvmti = NULL;
static jvmtiCapabilities capa;
jvmtiError error;

memset(&capa, 0, sizeof(jvmtiCapabilities));
capa.can_generate_method_entry_events = 1;
capa.can_generate_exception_events = 1;
capa.can_generate_vm_object_alloc_events = 1;

error = (*jvmti)->AddCapabilities(jvmti, &capa);
test_jvmti_error(jvmti, error, "Unable to get necessary JVMTI
capabilities.");
```

5. Next, you must enable notification of the events in Agent_OnLoad(); (for example, JVM initialization and JVM death events):

```
error = (*jvmti)->SetEventNotificationMode
(jvmti, JVMTI_ENABLE, JVMTI_EVENT_VM_INIT, (jthread)NULL);
error = (*jvmti)->SetEventNotificationMode
(jvmti, JVMTI_ENABLE, JVMTI_EVENT_VM_DEATH, (jthread)NULL);
test_jvmti_error(jvmti, error, "Cannot set jvmti callbacks");
```

6. After registering for the event notification, you must register callback functions for each event:

```
jvmtiEventCallbacks callbacks;

(void)memset(&callbacks, 0, sizeof(callbacks));
callbacks.VMInit = &callbackVMInit; /* JVMTI_EVENT_VM_INIT */
callbacks.VMDeath = &callbackVMDeath; /* JVMTI_EVENT_VM_DEATH */

error = (*jvmti)->SetEventCallbacks(jvmti,
&callbacks, (jint)sizeof(callbacks));
test_jvmti_error(jvmti, error, "Cannot set jvmti callbacks");
```

7. Then set the global agent data to use throughout the agent:

```

typedef struct {
    jvmtiEnv *jvmti;
    jboolean vm_is_started;
    jrawMonitorID lock;
} GlobalAgentData;
static GlobalAgentData *gdata;

```

8. Perform the following setup in Agent_OnLoad():

```

static GlobalAgentData data;
memset((void*)&data, 0, sizeof(data));
gdata = &data;
/* Here save the jvmtiEnv* for Agent_OnUnload().*/
gdata->jvmti = jvmti;

```

9. Create the raw monitor in the Agent_OnLoad to protect critical sections of the code (such as JVM initialization and JVM death event processing code).

Use enter_critical_section() to enter, and use exit_critical_section() to exit exit_critical_section(), as shown here:

```

error = (*jvmti)->CreateRawMonitor(jvmti, "agent data",
    &(gdata->lock));

static void enter_critical_section(jvmtiEnv *jvmti) {
    jvmtiError error;
    error = (*jvmti)->RawMonitorEnter(jvmti, gdata->lock);
    test_jvmti_error(jvmti, error, "Cannot enter with raw
monitor");
}

static void exit_critical_section(jvmtiEnv *jvmti) {
    jvmtiError error;
    error = (*jvmti)->RawMonitorExit(jvmti, gdata->lock);
    test_jvmti_error(jvmti, error, "Cannot exit with raw
monitor");
}

```

10. Example 9-1 on page 143 is a simple Java program (JVMTITest.java). Its main thread creates two threads. After the JVM is initialized, the agent receives a JVM initialization event. When the JVM exits, the agent receives a JVM death event.

```
public class JVMTITest {

    static MyThread t;

    public static void main(String args[]) {

        t = new MyThread();
        System.out.println("Creating and running 2 threads...");

        for(int i = 0; i < 2; i++) {
            Thread thr = new Thread(t,"MyThread"+i);
            thr.start();

            try {
                thr.join();
            } catch (Throwable t) {
            }
        }
    }

    class MyThread implements Runnable {

        Thread t;

        public MyThread() {
        }

        public void run() {
            try {
                Thread.sleep(1000);
            } catch (java.lang.InterruptedException e){
                e.printStackTrace();
            } catch (Throwable t) {
            }
        }
    }
}
```

11. Here are the steps to compile and execute the sample code:

- a. Export the PATH to the JDK and LIBPATH to point to the Agent location:

```
export PATH=/usr/lpp/java/J5.0/bin:$PATH
export LIBPATH=./:$LIBPATH
```

b. Generate the object file:

```
cc -c -o JVMTIAgent.o -W c,exportall
-I/usr/lpp/java/J5.0/include JVMTIAgent.c
```

c. Build the shared library:

```
cc -o libTIAgent.so -W l,dll JVMTIAgent.o
```

d. Compile the Java simple test case:

```
javac JVMTITest.java
```

e. When users execute the agent, they will receive the following output:

```
$ java -agentlib:TIAgent JVMTITest
```

```
Agent loaded
Got VM init event
Creating and running 2 threads...
Got VM Death event
Agent Unloaded
```

For more information about writing agents using JVMTI, refer to the following sites:

<http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html>

<http://java.sun.com/developer/technicalArticles/Programming/jvmti>

9.3 Transition from JVMPI to JVMTI

The Java Virtual Machine Profiler Interface (JVMPI) is available in the SDK 5.0, but the JVMPI has been deprecated. Users are encouraged to move to JVMTI.

Because the complexity to move from JVMPI to JVMTI depends on how JVMPI has been used, this section provides tips to help you make the transition successfully:

- The JVMTI offers more functionality and improves performance, as compared to JVMPI. Table 9-1 on page 145 is a simple list of JVMPI-to-JVMTI mapping.

Table 9-1 Mapping between JVMPI and JVMTI functions

| JVMPI | JVMTI |
|----------------------|---|
| GetCallTrace | GetStackTrace |
| EnableEvent | SetEventNotificationMode Use <code>jvmtiEventMode == JVMTI_ENABLE</code> |
| DisableEvent | SetEventNotificationMode Use <code>jvmtiEventMode == JVMTI_DISABLE</code> |
| DisableGC & EnableGC | There is no JVMTI equivalent because JVMTI does not require GC to be disabled for any events. |
| GetThreadStatus | GetThreadState |

- In the JVMTI, each event has its own callback routine. Table 9-2 shows the mapping between some of the events.

Table 9-2 Mapping between JVMPI and JVMTI events

| JVMPI | JVMTI |
|---------------------------|--------------------------|
| JVMPI_EVENT_METHOD_ENTRY | JVMTI_EVENT_METHOD_ENTRY |
| JVMPI_EVENT_METHOD_EXIT | JVMTI_EVENT_METHOD_EXIT |
| JVMPI_EVENT_JVM_INIT_DONE | JVMTI_EVENT_VM_INIT |
| JVMPI_EVENT_THREAD_START | JVMTI_EVENT_THREAD_START |
| JVMPI_EVENT_JVM_SHUT_DOWN | JVMTI_EVENT_VM_DEATH |

- The most important difference between JVMPI and JVMTI is the way Java heap is handled, as described here:
 - When you request heap dump using JVMPI, a single chunk of data is returned, which needs to be parsed by the agent.
 - In JVMTI, there are various functions available to the user to traverse the heap.

For more detailed information about the transition from JVMPI to JVMTI, refer to the following site:

<http://java.sun.com/developer/technicalArticles/Programming/jvmpitransition/>

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

<ftp://www.redbooks.ibm.com/redbooks/SG247291>

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG247291.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

| <i>File name</i> | <i>Description</i> |
|----------------------------|--|
| JZOS Job mgmt.zip | Samples for Chapter 3, “Job management using JZOS” on page 25 |
| JZOS Batch tool.zip | Samples for Chapter 4, “Java development and job management with Eclipse” on page 49 |
| JNI ASCII.zip | JNI samples in ASCII for 6.3, “More advanced JNI usage” on page 97 |
| JNI EC EBCDIC.zip | JNI samples in EBCDIC for 6.4, “Integrating Java and COBOL using IBM Enterprise Object-Oriented COBOL” on page 105 |
| JZOS Toolkit API | Samples for Chapter 7, “I/O using the JZOS toolkit API” on page 111 |
| JVMTI | Samples for Chapter 9, “Java Virtual Machine Tool Interface” on page 139 |

How to use the Web material

Download the .zip file you would like to use and unzip it at the workstation. Depending on the topics certain components may be for usage on the workstation and certain components may be for usage on z/OS. Use the files in the zip files as explained in the corresponding chapter.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 150. Note that some of the documents referenced here may be available in softcopy only.

- ▶ *Java Stand-alone Applications on z/OS, Volume I*, SG24-7177

Other publications

These publications are also relevant as further information sources:

- ▶ *z/OS 64-bit C/C++ and Java Programming Environment*, REDP-9110
- ▶ *IBM 31-bit and 64-bit SDKs for z/OS, Java 2 Technology Edition, Version 5SDK and Runtime Environment User Guide*, sdkguide.zos
- ▶ *z/OS C/C++ Programming Guide*, SC09-4765-06
- ▶ *Enterprise COBOL for z/OS Programming Guide*, SC27-1412
- ▶ *IBM WebSphere Developer for zSeries Host Planning Guide*, SC31-6599-03
- ▶ *MVS Programming: Assembler Services Guide*, SA22-7605-06
- ▶ *z/OS V1R7.0 MVS Programming: Authorized Assembler Services Guide*, SA22-7608-09
- ▶ *MVS System Management Facilities (SMF)*, SA22-7630-12
- ▶ *z/OS C/C++ Run-Time Library Reference*, SA22-7821-07
- ▶ *UNIX System Services Programming: Assembler Callable Services Reference*, SA22-7803-08
- ▶ *z/OS V1R7.0 Using REXX and z/OS UNIX System Services*, SA-7806-08
- ▶ *IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide*, SC34-6650

Online resources

These Web sites and URLs are also relevant as further information sources:

- ▶ IBM 64-bit SDK for z/OS Java 2 Technology Edition, V5:
<http://www-03.ibm.com/servers/eserver/zseries/software/java/j5pcont64.html>
- ▶ Java on z/OS reference information:
<http://www-03.ibm.com/servers/eserver/zseries/software/java/javaintr.html>

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

Numerics

64-bit support in z/OS V1R6 92

A

Ant Buildfile 51
Ant FTP support 52
Assembler Callable Services 86
ASSIZEMAX parameter 12

B

BPX.SMF FACILITY 98
BPXBATCH 10, 26
BPXBATCH, REGION parameter 12
BPXBATSL 10, 26
BPXWDYN 118

C

C/C++ 92
class cache 37
 size 38
class sharing, cache size 138
class sharing, enabling 138
COBOL 92
CPU problems 135
crash problems 132

D

deadlock 131
deadlock problems 131
debugging options 67
debugging, options in JVM 68
delopy.xml file 58
Diagnostic Tooling Framework for Java (DTFJ)
129, 135
DTFJ (Diagnostic Tooling Framework for Java)
129, 135
Dump Formatter 134

E

Enterprise Generation Language (EGL) 72
Entry-Sequenced Data Set (ESDS) 120

F

Fortran 92

G

Garbage Collection 13
 compaction 15
 compaction phase 20
 compaction, conditions 21
 concurrent mark 17
 concurrent sweep 19
 heap expansion 22
 heap expansion amount 23
 heap shrinkage 23
 mark 15
 mark stack 16
 mark stack overflow 16
 parallel bitwise sweep 19
 parallel mark 17
 PhantomReference 21
 reference object 21
 SoftReference 21
 sweep 15
 sweep phase 18
 tracing 16
 WeakReference 21
 work packets 16
Garbage Collection (GC) policies 14
 gencon 15
 JVM parameter to specify 15
 optavgpause 14
 optthruput 14
 subpool 15
Garbage Collector 11
GC policy 136
gencon 136

H

hang problems 131
heap size 13
heapdump 131
High Level Language (HLL) 92
HLL 92

I

IBM Enterprise COBOL for z/OS 106
 compile and link commands 108
 compiling COBOL 106
 compiling, output files generated 107
 compiling, source with OO syntax 106
 example, COBOL program calling Java 109
 example, Java program calling COBOL 108
 JNI.cpy file 107
 linking 107
 linking, DLL naming convention 107
 linking, DLL naming convention using a Java
 class which is part of a package 107
 linking, DLL side files required 107
IEFUSI 12
igzjava.x 107
Interactive Problem Control System (IPCS) 133
Inter-Language Communication (ILC) 101
IPCS (Interactive Problem Control System) 133

J

J2EE 87
Java 2 Enterprise Edition (J2EE) 6
Java class sharing, enabling 38
Java dump, generating 131
Java Native Interface 85–86, 93
Java Process ID (PID) 131
Java Record I/O library(JRIO) 112
Java Software Development Kit (SDK), available
 products on z/OS 7
Java Virtual Machine 86
Java Virtual Machine Debug Interface (JVMDI) 140
Java Virtual Machine Profiler Interface (JVMPPI) 139
Java Virtual Machine Tool Interface (JVMTI) 139
 example 140
 loading agents 140
jdumpview utility 134
jextract utility 134
JIT compiler, disabling 133
JNI 85–86, 93
 jni.h header file 89
 object 89
 mapping of primitive types 89
 object types 89
 primitive types 88
 reference types 88–89
JNI weak references 22
Just In Time Compiler (JIT) 6, 138

Just In Time compiler, performance functions 138
JVM 86
JVM parameter, for specifying GC policy 15
JVMDI (Java Virtual Machine Debug Interface) 140
JVMPPI (Java Virtual Machine Profiler Interface) 139
JVMTI (Java Virtual Machine Tool Interface) 139
JZOS
 classes 113
 com.ibm.jzosZUtil 45
 MvsCommandCallback 45
 registerMvsCommandCallback 45
 compared to BPXBATCH and BPXBATSL 27
 data sets supported 112
 debugging
 IBM_JAVA_OPTIONS 82
 environment variables 28
 JZOS_ENABLE_MVS_COMMANDS 29
 JZOS_ENABLE_OUTPUT_TRANSCODIN
 G 29
 JZOS_GENERATE_SYSTEM_EXIT 29
 JZOS_MAIN_ARGS 29
 JZOS_MAIN_ARGS_DD 29
 JZOS_OUTPUT_ENCODING 29
 logging levels 30
 methods
 getCurrentJobname 45
 getCurrentUser() 45
 getDefaultPlatformEncoding() 45
 getEnv() 45
 getJavaVersionInfo() 45
 methods for WTO 44
 return code, examples 36
 return codes 29
 toolkit library 112
 ZFile methods 113
 ZFile methods used for VSAM access 122
JZOS, functions 26

K

Key-Sequenced Data Set (KSDS) 120
KSDS cluster 120

L

Language Environment (LE), 64-bit support 92
Large Object Area (LOA) 137
LDS (Linear Data Set) 120
LE (Language Environment) 92
libjvm.x 107

Linear Data Set (LDS) 120
LISTUSER command 11
log4j 38
Long Pointer 64 92
LP64 92
lpex 72
LRECL 121

M

MAXASSIZE parameter 12
-memorycheck JVM option 130
MVS modify command 43
MVS stop command 43

O

OMVS segment, displaying information 12
Open Group consortium 92
optavgpause 20, 137
optthruput 137
OutOfMemory problems 130

P

performance problems 135
PL/I 92

R

Redbooks Web site 150
 Contact us xi
region size 11
region size, checking 11
region size, for telnet and SSH users 12
Relative Record Data Set (RRDS) 120
Remote Job Monitor 80
Resource Management Facility (RMF) 135
RMF (Resource Management Facility) 135
RRDS (Relative Record Data Set) 120
RTL (Run Time Library) 86
Run Time Library (RTL) 86

S

Service Flow Modeler (SFM) 72
SFM (Service Flow Modeler) 72
Sharing of classes 37
Small Object Area (SOA) 137
smf_record callable service 98
SOA (Small Object Area) 137
stack trace 133

subpool 137
System Display and Search Facility (SDSF) 42
System Management Facility (SMF) 98
System.exit() 36
System.gc() 15

T

transaction dump, tools to analyze 133

U

utility
 jdmpview 134
 jextract 134

V

verbosegc 14, 130
Virtual Sequential Access Method (VSAM) 120

W

WDz (WebSphere Developer for zSeries) 72
Websphere Application Server 87
WebSphere Developer for zSeries (WDz) 72
 connection setup 77
 filters 78
 FMIDs required 76
 host requirements 75
 workstation requirements 74
Write-To-Operator (WTO) 42

X

X/Open Portability Guide (XPG) 8
-Xdebug 68
-Xgcpolicy 20
Xgcpolicy 136
-Xgcthreads 17
-Xint 135
-Xloainitial 137
Xloamaximum 137
-Xmaxt 23
-Xminf 23
-Xms 13
Xms 131
-Xmx 13
Xmx 130–131
-Xnocompactgc 21
XPG 4.2 standard 8
-Xquickstart 135

-Xrunjdpw 68
-Xrunjnichk 130
-Xsoftrefthreshold 22

Z

z Application Assist Processor (zAAP) 87
z/OS Language Environment (LE) 8
z/OS UNIX Application Services 8
z/OS UNIX System Services 8
zAAP (z Application Assist Processor) 87
zos.properties 57
zos.properties, debugging option 68
zSeries Application Assist Processor 7

Java Stand-alone Applications on z/OS Volume II

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



Redbooks

Java Stand-alone Applications on z/OS Volume II

Using JZOS to develop, deploy and run Java applications

This IBM Redbook is the second redbook in a series of two about Java stand-alone applications on z/OS. We recommend using this document as a complement to *Java stand-alone Applications on z/OS Volume I*, SG24-7177.

Exploring the Java SDK 5

This book explains some topics discussed in Volume 1 in more depth, and also provides information about additional topics. It pays special attention to JZOS, a newly integrated function in the z/OS SDKs that can be used to run Java batch jobs and perform I/O

Java Native Interface

This document also features other interesting topics, such as the use of Eclipse and WebSphere Developer for zSeries for integrated development of stand-alone Java applications, Java Native Interface (JNI), the new Java Virtual Machine Tool Interface (JVMTI) and problem determination with SDK 5 on z/OS.

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