

Java Stand-alone Applications on z/OS Volume 1

Setting up an environment for
managing Java programs on z/OS

Building, deploying, running, and
debugging applications

Analyzing application
performance and
exploiting zAAP



Alex Louwe Kooijmans
Paul Anderson
Kenichi Yoshimura
Linfeng Yu



International Technical Support Organization

**Java Stand-alone Applications on z/OS
Volume 1**

May 2006

Archived

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

First Edition (May 2006)

This edition applies to IBM SDK for z/OS, Java 2 Technology Edition, productnumber 5655-I56.

© 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	x
Become a published author	xi
Comments welcome	xi
Chapter 1. Scenarios for running Java stand-alone applications on z/OS	1
1.1 Definition of Java stand-alone applications	2
1.1.1 Definition of batch applications	2
1.2 Legacy system modernization	3
1.3 Reasons to use Java for stand-alone applications	5
1.3.1 Functionality	6
1.3.2 XML processing	6
1.3.3 Reusing J2EE online program logic	7
1.3.4 Migrating Java applications for QoS and server consolidation	7
1.3.5 Availability of skill set for the future	8
1.3.6 zAAP processor	8
1.4 Topologies of Java stand-alone applications	9
Chapter 2. Job management	11
2.1 The first Java program on z/OS	12
2.2 Running a Java program as a batch job	12
2.2.1 Overview of Job Control Language	13
2.2.2 Writing JCL for a Java batch job	18
2.2.3 Limitations of BPXBATCH	20
2.2.4 BPXBATSL	21
2.2.5 Comparison of JCLBATCH and JCLBATSL	22
2.3 Developing multi-step jobs	22
2.3.1 Specifying multiple steps in a JCL script	23
2.3.2 Conditions on a step	25
2.3.3 Conditional step execution using IF-ELSE	26
2.4 Design considerations for multi-step jobs	27
2.4.1 What skill sets are available in the development team?	27
2.4.2 How much integration with legacy applications is necessary?	27
2.4.3 How much control is required between steps?	27
2.4.4 Return code from BPXBATCH and BPXBATSL	28

Chapter 3. Access to MVS data sets	31
3.1 The JRIO library	32
3.1.1 JRIO classes and interfaces	32
3.1.2 JRIO usage examples	35
3.2 Working with VSAM data sets	41
3.2.1 VSAM overview	41
3.2.2 Accessing records in a VSAM data set using the JRIO library	42
3.3 Working with COBOL packed decimal numbers	46
3.3.1 Converting from COBOL packed decimals to Java integers	48
3.3.2 Converting from Java integers to COBOL packed decimals	49
3.4 JRecord bean generator and supporting APIs	50
3.4.1 JRecord utility APIs	50
3.4.2 Generating a Java bean for a record	50
3.4.3 JRecord runtime library and plug-in installation	60
Chapter 4. Tools and techniques	61
4.1 Introduction	62
4.2 Application build and deployment	62
4.2.1 Ant support in Eclipse	62
4.2.2 Set up Ant ftp support in Eclipse	65
4.2.3 Build and deployment Ant script	66
4.2.4 Use the ANT buildfile	70
4.2.5 Further discussion	71
4.3 Debugging	72
4.3.1 Remote debugging	72
4.3.2 Preparing the Java program	73
4.3.3 Creating a shell script and JCL	75
4.3.4 Deploying and starting a Java application	78
4.3.5 Work with the debugger in the Eclipse workbench	82
4.3.6 Local debugging	89
4.3.7 Rational Application Developer considerations	90
Chapter 5. Measuring performance and application profiling	91
5.1 Performance analysis	92
5.1.1 Tooling overview	92
5.1.2 Establishing a goal	93
5.2 Overview of analysis strategies	94
5.2.1 Statistical	94
5.2.2 Fractional	94
5.2.3 Elimination	95
5.2.4 Accurate	95
5.3 Statistical analysis	95
5.4 Fractional analysis	96

5.4.1	TIMEUSED	103
5.5	Elimination analysis	111
5.6	Accurate analysis	112
5.6.1	Installing TPTP	112
5.6.2	Profiling an application on z/OS	126
5.6.3	Fractional analysis with TPTP	155
5.7	Code listings	165
Chapter 6.	Exploiting zAAP	173
6.1	zAAP prerequisites	174
6.2	Overview of zAAP and runtime options	174
6.3	Exploring zAAP utilization potentials	175
6.3.1	zAAP utilization estimate and projection tool	175
6.3.2	Experimental results of zAAP utilization estimates	178
6.3.3	Hardware and software configuration	181
Chapter 7.	Problem determination	183
7.1	Introduction	184
7.2	Checking the z/OS environment	184
7.2.1	Maintenance	184
7.2.2	LE settings	185
7.2.3	Environment variables	185
7.2.4	Private storage usage	186
7.2.5	Service Class for the application	186
7.3	Collecting data for problem determination	187
7.3.1	JVM dump control	188
7.3.2	CEEDUMP, SYSTDUMP and SVCDUMP	189
7.3.3	JAVADUMP and HEAPDUMP	191
7.3.4	Other problem determination data	192
7.4	Dump analysis and tools	193
7.4.1	Analyzing SYSTDUMP and SVCDUMP	194
7.4.2	Interpreting a JAVADUMP	199
7.4.3	HeapAnalyzer	206
7.4.4	Other analysis tools	214
7.5	Problem determination scenarios	214
7.5.1	Diagnose crashes	214
7.5.2	Debugging hangs	214
7.5.3	Debugging memory leaks	214
7.5.4	Debugging performance problems	215
Appendix A.	Additional material	217
	Locating the Web material	217
	Using the Web material	217
	System requirements for downloading the Web material	218

Related publications	219
IBM Redbooks	219
Other publications	219
Online resources	220
How to get IBM Redbooks	220
Help from IBM	221
Index	223

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

Trademarks

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

@server®

@server®

Redbooks (logo) ™

eServer™

z/OS®

zSeries®

z9™

CICS®

DB2 Connect™

DB2®

Iterations®

IBM®

IMS™

Language Environment®

MVS™

Rational®

Redbooks™

RACF®

System z™

System z9™

VisualAge®

WebSphere®

The following terms are trademarks of other companies:

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

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

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

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 about using Java™ on z/OS® to develop, deploy, and run stand-alone and batch applications.

Java has certainly become the most popular programming language, and it is the language of choice for many enterprise applications today. This is the result of many advancements in the language itself as well as related technologies and tools that enable users to develop and run cost-effective, scalable, high-performance solutions.

IBM System z™¹ servers provide comprehensive support for deploying Java applications while providing the traditional quality of service, advanced security, and scalability that users have come to expect from this server family. IBM System z Application Assist Processors (zAAP)² further enhance the ability of an enterprise to implement complex and heavy workload applications on System z in a cost-effective and reliable manner.

There are many technical and business-oriented documents that describe the values of running J2EE™ applications on System z using WebSphere® Application Server. However, less attention has been given to running stand-alone Java applications. We envision Java stand-alone applications to be typically one of the following types:

- ▶ A batch application with the executed program written in the Java language and run by using a JCL and typically started by a workload scheduler
- ▶ An application started from the UNIX® command line, either continuously running or started and stopped at intervals

This book is a technical guideline for Java application developers who do not have prior experience with the System z operating environment, as well as System z operators who are new to the world of Java because of the inevitable transition occurring in the IT industry. Chapters in this book are designed to be self-contained so that they can be read in any order.

¹ IBM System z servers encompass both the IBM zSeries® models and the System z9™ models. Throughout this document we use the term “IBM System z.”

² Refer to Chapter 6, “Exploiting zAAP” on page 173 for more details and prerequisites.

The team that wrote this redbook

This 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 at the International Technical Support Organization, Poughkeepsie Center. He leads residencies to create Redbooks™ in the areas of Java and WebSphere on z/OS as well as teaching workshops on these topics worldwide. Alex has worked for IBM since 1986. Prior to joining the ITSO, he worked in various other roles, including IT Specialist supporting customers in Europe getting started with WebSphere and Client IT Architect in the financial services sector in The Netherlands. His areas of expertise include Java and WebSphere on z/OS, and integrating WebSphere with DB2®, MQ, CICS® and IMS™.

Paul Anderson has worked in the IT industry for more than 20 years, including eight years for IBM. He has experience in many aspects of development and operations, and is an expert in solving problems related to performance issues. Paul's areas of expertise include Java, c, c++, RPG and assembler on a variety of platforms, and his development experience ranges from large business applications to low-level device programming. Much of his focus in the last few years has been helping customers, ISVs, and IBM itself to deal with Java and System z performance issues.

Kenichi Yoshimura is a software engineer in the Australian Programming Center, IBM Software Group in Perth, Australia. Since joining IBM in 2004, he has worked on several zSeries software products including Fault Analyzer for z/OS, SCLM Developer Toolkit, and VisualAge® Generator EGL Plug-in for VSE. He holds a Master Degree in Engineering Science from the University of Melbourne.

Linfeng Yu is an architect with Insurance Services Office, Inc. He has extensive experience in developing large-scale, complex enterprise-wide architectures and cross-platform software development. He has worked with Java and WebSphere for z/OS for more than five years.

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

Richard Conway
International Technical Support Organization, Poughkeepsie Center

Norman Aaronson, Clarence Clark, John Rankin
IBM Poughkeepsie lab, Java for z/OS

Donald Houghtalen
IBM Endicott, Java for z/OS

Adrian Simcock, Liam Doherty
Australian Programming Center, IBM Software Group in Perth, Australia

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 e-mail to:

redbook@us.ibm.com

- Mail your comments to:

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



Scenarios for running Java stand-alone applications on z/OS

In this chapter, we define what we consider as Java stand-alone applications on z/OS and describe scenarios where Java-based solutions should be considered. We also discuss the placement and offerings of Java stand-alone applications from the legacy system modernization process view.

1.1 Definition of Java stand-alone applications

We define a Java stand-alone application as an application that it is not running inside an application server. In technical terms, a Java stand-alone application has a *main* method, which is similar to a main entry point of a traditional COBOL or PL/1 load module. The main method is the first method to be executed when the application starts and may have parameters fed to the application. Another characteristic is that a Java stand-alone application runs in its own Java virtual machine (JVM™), which is actually started first, before the first statements of the application are executed. Finally, the Java functionality available to a Java stand-alone application is not quite the same as the J2EE programming model offered by a typical J2EE application server, such as WebSphere Application Server. An important example is the use of the Java Native Interface (JNI), which is very common in stand-alone applications, but not recommended inside an application server. Furthermore, JNI is definitely not part of the J2EE programming model. In fact, it is the other way around: the J2EE programming model offers Java components that need a *container* to execute. Enterprise JavaBeans™ (EJBs) run in the EJB™ container of the J2EE application server.

Having said this, we now know that a Java stand-alone application is autonomous and needs a trigger to be launched. This trigger can be:

- ▶ Somebody manually entering a command on the UNIX command line to start a JVM with the application
- ▶ A job scheduler starting a JCL that starts a JVM with the application

Note that we are only talking about a triggering mechanism at this point, and *not* about the length of time a Java stand-alone application is active. A Java stand-alone application can be running between a fraction of a second and, theoretically, as long as the system stays powered on. Most pure Java application servers and http servers can be considered Java stand-alone applications that have been started once and stay alive as long as the system holds out. They listen for incoming requests and process the requests when they come in. If nothing comes in, the application goes to sleep.

1.1.1 Definition of batch applications

Stand-alone applications and batch applications are two different things. A stand-alone application may have an on-line character and may not be performing repetitive tasks at all. The way it is launched, however, could be the same way a real batch job is launched. Also, a stand-alone application may start and finish during on-line hours and the output may be required immediately.

This brings us to the question: What is batch? We can call something a batch job if it meets the following criteria:

- ▶ Start and stop time are not critical in terms of seconds or even minutes. There is no terminal or browser waiting for the output of the batch job.
- ▶ The workload executed by the batch job is typically a repetition of the same task, sometimes up to millions of times. Processing efficiency can be achieved in this way.

A typical example of a batch job is the monthly printing of account balance statements. In the real world this process alone would consist of many different batch jobs. If we printed each account balance statement at the end of the month as an OLTP transaction, the process would never be able to finish within a night, a day, or even a week!

A batch job is a unit of work that is performed in sequence, and its execution is controlled by the operating system based on the job's priority and resource requirements. In Chapter 2, "Job management" on page 11, we discuss how to describe a batch job and manage steps within a job using Job Control Language (JCL). Like any other mainframe application, a Java batch application's job requirements must be specified in a JCL script. Once such a Java application is submitted to the system, it is the responsibility of the operating system and supporting subsystems to control and manage the execution of the application. The operating system and supporting subsystems ensure that required resources are allocated for the job, and specify when and where to run the job transparently to the application. For example, if zAAP processors are installed, it delegates the execution of Java application instructions to zAAP processors.

1.2 Legacy system modernization

All applications go through three different phases during their lifetime: *maintenance*, *modernization* and *replacement* (Comella-Dorda et al. 2000). These phases are inevitable because the requirements for a business change more rapidly than ever. Maintenance is a continuous process of making small changes to an existing application to customize, optimize performance, and fix problems to meet the changing requirements of the business.

The modernization and replacement phases of an application are particularly relevant to stand-alone Java applications on System z. While an application can be deployed for a long time with continuous maintenance, it eventually gets to a stage where it is no longer acceptable in the original format of the application. This might occur because of obsolete user interfaces or incompatible operating environments. The first stage of a modernization process typically involves

implementation of a wrapping layer that provides newer interfaces using techniques such as *screen scraping*.

Such an approach offers many advantages, including the following:

- ▶ There are tools that facilitate transformation of existing applications
- ▶ This approach protects the investments of the past (that is, in existing applications) and it is much less risky than replacement with a newer system.

However, such solutions are typically hard to maintain because this technique does not modernize the underlying applications themselves.

The last phase of a legacy system modernization process is replacement. In this case a legacy system is completely replaced with a new system. This phase involves much greater risk because there is no guarantee that the new system will provide better service to its users, even with the best intentions and design for the new application. Do not underestimate the value of legacy applications; they have been supporting the business successfully for many years.

Once a company is committed to replacing a legacy application, several factors must be considered:

- ▶ Availability of skills for the future
Many batch applications have typically been written in COBOL for mainframe platforms. Although the concern that COBOL language skills would disappear has been proven wrong, it is still valid to say that the number of COBOL developers is decreasing.
- ▶ Integration
Enterprise applications no longer exist in isolation. They are typically exposed to other systems over the network (for example, the systems of other departments, business partners, and customers). Modern programming languages and tools typically offer much easier integration frameworks, which leads to more cost-effective development of new applications. Application development can focus on the real business logic rather than on the glue to connect all the different pieces together.
- ▶ Performance
The expected performance of a new application may dictate how it can be re-implemented. A batch job typically processes a large volume of data within a certain time frame. In other words, the throughput of an application is an important factor.
- ▶ Portability
Low-level programming languages such as Assembler offer much better performance, but are typically not portable to any other platform. This brings up the questions: Will the new application always run only on System z? Does any other ready-made software serve the purpose with minor modifications?

Considering these factors, choosing the Java programming language for development of new applications offers several advantages. Table 1-1 summarizes the benefits of the Java programming language with respect to the identified criteria.

Table 1-1 Java programming language benefits

Criteria	Benefits of Java
Skill availability	Java developers are much easier to find today. Much of IT education is centered around object-oriented (OO) design, and the Java programming language is easily accepted by the newer generation of developers. In addition, there are many tools that improve productivity of developers in integrated development environments (IDEs) such as Eclipse. It is reasonable to assume that the newer generation of developers would prefer to work in this kind of environment rather than working in a green screen environment.
Integration	Java is particularly well-suited to integration efforts because of the availability of a wide range of APIs (for example, JDBC™), XML support, and easy to use network communication interfaces, just to name a few attractive features of the language.
Performance	The performance of Java applications has improved over the years because of advancements of the language and tools such as the Just-in-Time (JIT) compiler. Although the performance of Java may not match simpler run-time languages such as Assembler and COBOL yet, the difference may be small enough to not be a decisive factor anymore. We discuss the performance of Java applications in z/OS and approaches to performance tuning in Chapter 5, “Measuring performance and application profiling” on page 91.
Portability	The “ <i>write once, run anywhere</i> ” characteristic of the Java programming language offers a significant advantage over any other language, and there are many ISV products developed in Java because of this feature. In many cases, existing applications can be ported to z/OS with minimal modifications. Even when some resources are not accessible natively in Java, programs written in native languages can be included in a Java application easily using Java Native Interface (JNI).

1.3 Reasons to use Java for stand-alone applications

In this section, we describe scenarios where Java could be considered for development of stand-alone applications on z/OS.

1.3.1 Functionality

It will not take a lot of time to find out that most modern application requirements can be tackled more easily in Java than in any other language. Numerous reusable objects already exist. These objects perform a wide variety of functions, from a simple date conversion to a full class library doing graphics.

Although graphics will not help us a lot in a batch application on z/OS, consider requirement such as sending an e-mail as a confirmation that a database record has been updated, or updating a remote Oracle database from within a batch job on z/OS. Both examples can be implemented with Java without any complex coding, while with COBOL this would require a very challenging effort.

1.3.2 XML processing

The Extensible Markup Language (XML) is the foundation of many integration solutions today because of its flexibility and extensibility for describing data, and its support for simple transmission of data over the network using protocols such as Hyper Text Transfer Protocol (HTTP). For example, service-oriented architecture (SOA) is entirely based on XML standards, including the simple object access protocol (SOAP) for the transport layer, Web service description language (WSDL) for service description, and Universal Description, Discovery and Integration (UDDI) for service discovery. These standards are intended to be acceptable frameworks across the industrial boundaries of the future, so they will likely touch most of the existing applications sooner or later. There are additional industry-specific standards, such as Financial Information Exchange protocol (FIX), Financial product Markup Language (FpML) and ISO 15022XML for the finance industry, and XML-based standards proposed by the Standards for Technology in Automotive Retail (STAR) group for the automotive retail industry. XML-based integration solutions enable integration of applications within an organization, and with customers and business partners.

Although many programming languages offer support for XML documents, Java is the language of choice to develop a new application with XML today, mainly because of the availability of APIs to work with XML documents such as SAX and DOM parsers, as well as readily available libraries for different protocols. In the era of “open standards,” platform-neutral languages such as Java have the best chance of finding the latest and greatest libraries for newer protocols and standards¹. The availability of such libraries facilitates a cost-effective development of prototypes and new applications.

¹ It is often dangerous to include open source/free libraries in an application because their performance and reliability are often not tested extensively. This is especially true for System z applications because their performance and running cost of applications are closely monitored.

1.3.3 Reusing J2EE online program logic

J2EE applications designed to run in application servers such as WebSphere Application Server are increasingly becoming more complex. The latest advancements in application server technology provide frameworks to deploy complex applications reliably and handle heavy workloads. For example, WebSphere Extended Deployment (XD) provides a framework for high performance and heavy workload applications, and added manageability of such applications.

Just as these advancements enable deployment of complex and heavy workload applications to application servers, the possibility of running such applications as stand-alone Java batch applications should be considered as well; the ability and manageability of complex and heavy workload applications are the key strengths of batch jobs on System z. Batch job solutions have a proven record of reliability and manageability over many years. Such migration also enables easier integration with other existing jobs in the system because all the jobs are controlled and managed by JCL scripts. In “Developing multi-step jobs” on page 22, we describe how to invoke a Java application as a job step and integrate with other programs in a JCL script.

1.3.4 Migrating Java applications for QoS and server consolidation

IBM System z servers are characterized by a level of stability, security, resiliency, and scalability that no other platform can offer. When these characteristics are the highest priorities of an application, System z is the platform of choice in many situations today. The following Web site presents some of the success stories about the Quality of Service achieved using the IBM System z platform:

<http://www-03.ibm.com/servers/eserver/zseries/success/>

Many years of advancements and evolution in information technology left businesses with complex IT infrastructures that are heterogeneous and distributed. In order to simplify such IT infrastructures, reduce the overall cost, and prepare for the future expansion of the business, many businesses choose to consolidate their IT infrastructures using IBM System z servers. The following Web site presents some examples of successful server consolidation with zSeries:

<http://www-03.ibm.com/servers/solutions/serverconsolidation/>

In these scenarios, existing applications are migrated to System z. Java applications are easy to migrate since the IBM Java SDK for z/OS is certified for Java compatibility. Hence, all Java applications which are written in the pure Java programming language should run on System z without any modification.

A typical problem in migration efforts occurs when the Java Native Interface (JNI) is used to implement some aspects of the applications. In many cases, re-implementation of the native code is feasible, and it may even be possible to find a better solution using mainframe resources such as data sets that offer efficient record-oriented data access natively. We discuss how to use the Java Record I/O (JRIO) library, which is a zSeries-unique I/O library, to access mainframe file systems in Chapter 3, “Access to MVS data sets” on page 31.

1.3.5 Availability of skill set for the future

The argument of disappearing COBOL applications has been proven wrong, and many businesses still heavily rely on COBOL applications today. However, the trend of transitioning away from such development tools on the mainframe continues inevitably. There are several reasons for this trend:

- ▶ The Java programming language has been one of the most popular programming languages over the last several years. Therefore, it is easier to find developers with Java development skills than to find COBOL developers.
- ▶ A wide range of libraries and APIs are available for the Java programming language. These libraries and APIs facilitate efficient development of prototypes and new applications. Since requirements and environments of businesses change rapidly, and new standards and protocols are introduced frequently, the availability of libraries and APIs offers significant advantages.
- ▶ Java applications are typically developed in powerful integrated development environments (IDE) such as Eclipse, which improve the productivity of software development significantly. In addition to providing programming assistance, such environments typically provide GUI-based debugging and profiling tools, which help in problem determination and performance tuning. We discuss how to use such tools in Chapter 4, “Tools and techniques” on page 61.
- ▶ Current IT education centers around object-oriented design. Consequently, the new generation of developers are trained for tools and languages based on such methodology. Because they are accustomed to working in such an environment, they would prefer to develop applications in Java using powerful tools, rather than working in a green screen environment.

1.3.6 zAAP processor

The zAAP processor is a low-cost processor, both in terms of hardware cost and IBM software license cost. As discussed in “Exploiting zAAP” on page 173, a Java application can run up to 100% of its instructions on the zAAP processor, which is very attractive from a cost point of view. Running the same application in the COBOL language would have to be done entirely on a General Purpose processor.

1.4 Topologies of Java stand-alone applications

We already mentioned a few examples of Java stand-alone (batch) applications, but there is more. The following scenarios are good examples to consider Java:

- ▶ Data processing jobs
A large volume of data to be processed as a batch job, for example, VSAM data sets or DB2 data.
- ▶ Messaging
Using messaging to communicate with other environments. JMS can be used from within a stand-alone or batch application to feed a (remote) queue. This way an application can integrate with almost any other environment.
- ▶ Printing
A large volume of print work to be completed as a batch job.
- ▶ Print servers, terminal servers and HTTP servers
- ▶ ISV applications.

Java-based solutions offer advantages in many situations. However, we do not necessarily recommend Java-based solutions in every case. This is especially true in situations where:

- ▶ COBOL skills are readily available in an organization
- ▶ The absolute priority of an application is the performance

Job management

In this chapter we discuss the most important things to know about the run-time environment of a Java batch program on z/OS. We discuss the following topics:

- ▶ How to write and compile a Java program on z/OS
- ▶ An overview of JCL and the job execution environment
- ▶ How to run a Java program as a batch job
- ▶ BPXBATCH and BPXBATSL
- ▶ Integration of legacy and modern applications using JCL
- ▶ Job execution management using conditions in JCL scripts

2.1 The first Java program on z/OS

The “write once, run anywhere” characteristic of Java programs also applies to z/OS. The development steps are the same as on any other platform. These characteristics make a Java-based solutions attractive for many organizations because Java developers are easier to find and train now than they were in the past.

Example 2-1 A simple Java program

```
public class Example {  
    public static void main (String args[]) {  
        System.out.println("First Java Program on zSeries!");  
    }  
}
```

Example 2-1 shows a simple Java class called `Example`, which prints a message to the standard output. The source of the `Example` class is defined in a text file called `Example.java`. This class can be compiled into Java byte code using the **javac** command:

```
javac Example.java
```

Assuming there is no problem with the program, it produces the `Example.class` file. To execute the program, type the following command at the command prompt:

```
java Example
```

The program simply prints out the message and quits. In all cases, a Java program will rely on environmental setting such as the `CLASSPATH` variable so that the Java runtime environment can find the classes required. For further details about how to set up the run-time environment, refer to the Java documentation.

2.2 Running a Java program as a batch job

To run a Java application as a batch job, the following steps are required:

1. Write the Java program and compile it.
2. Prepare a JCL script to run the Java program.
3. Submit the job.
4. Examine the job output.

The first step was described briefly in the previous section and it should not present much difficulty for Java developers because the process of writing and compiling Java programs is the same on any platforms.

2.2.1 Overview of Job Control Language

Job Control Language (JCL) is a scripting language used to control the execution of programs in mainframe operating environments. A JCL script describes the unit of work to be performed (job steps), resource allocations (for example, data set and memory), and much more.

A JCL script typically consists of a job card, data definition (DD) statements, and EXEC statements. A *job card* marks the beginning of a job and tells the system how to process the job. It includes information such as a job name, accounting information, and message level and class.

An *EXEC* statement identifies a step within a job and describes a program to be executed in the step. A job typically consists of several job steps.

The *REGION* parameter specifies the virtual storage allocation required for this step. For example, *REGION=32M* means the step requires 32 megabytes of virtual storage.

The *COND* parameter is used to specify when this step should be executed. For example, *COND=(0,NE)* means this step gets executed only when all of the previous steps finished with a return code of 0.

DD statements describe the I/O resources required for the step.

The syntax of JCL scripts is very strict. It is especially sensitive regarding the starting positions of various keywords and the continuation of a statement on the next line. Basic JCL syntax is summarized below:

- ▶ All JCL statements must start in column 1 and be identified by // (two slashes) at the beginning of the line.
- ▶ All statements must start in column 1 and end in column 71.
- ▶ A comma indicates the statement has a continuation.
- ▶ A continuation of a statement must start between column 4 and 16.
- ▶ /* at the start of the line marks a comment line.

Tip: Since the syntax of JCL scripts is very strict, the syntax highlighting feature of the ISPF editor may help identify problems. The syntax highlighting feature of the ISPF editor can be turned on by typing the following command at the command prompt:

HIGHLIGHT ON

Once a job is submitted to the system, it is the responsibility of the *Job Entry Subsystem (JES)* to schedule and execute the job. There are two versions of JES systems supported, JES2 and JES3. An overview of the job submission and execution process is shown in Figure 2-1. A user submits a job to the system via JES. JES interprets the JCL and passes the job to the system for processing with required resources. Once the job is completed by the system, JES collects the output and information about the job and returns it to the user.

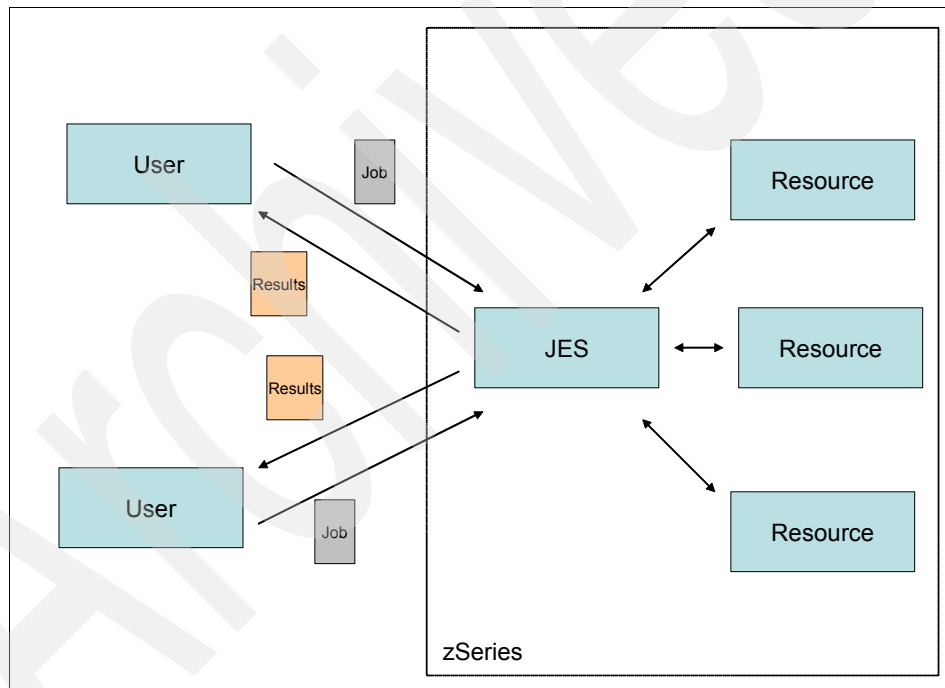


Figure 2-1 Overview of job submission and execution

Example 2-2 on page 15 shows a typical JCL script that invokes a REXX exec called MYREXX. REXX execs are invoked using a program called IRXJCL. IRXJCL must know where to find the REXX exec to execute, which is specified using a DD statement called SYSEXEC. The MYREXX exec uses two

resources, one for the input and another for the output. These are also specified using DD statements (INFILE and OUTFILE, respectively).

Example 2-2 JCL script to invoke a REXX exec

```
//REXXJOB JOB (ITS0),'ALEX02',REGION=30M,  
//      CLASS=A,MSGCLASS=X,NOTIFY=&SYSUID  
//MSG     EXEC PGM=IRXJCL,PARM='MYREXX'  
//SYSTSIN DD SYSOUT=*  
//SYSPRINT DD SYSOUT=*  
//INFILE  DD DSN=USER1.DATA.DATAIN,DISP=SHR  
//OUTFILE DD DSN=USER1.DATA(DATAOUT),DISP=SHR  
//SYSEXEC DD DSN=USER1.EXEC,DISP=SHR
```

The easiest way to submit a job to the system is to type the **submit** (or **sub**) command from the ISPF editor after you finish writing the JCL script. Alternatively, a job can be submitted from the ISPF Data set List Utility (option 3.4). Once the job is submitted successfully, it will display the following message:

JOB *jobname* (*jobnumber*) SUBMITTED

These approaches are typically *ad hoc* and may not be suitable for a production system. However, the discussion of more sophisticated job scheduling and submission is beyond the scope of this book.

Once the job is submitted to the system, the status of the job can be monitored and examined from the *System Display and Search Facility (SDSF)*. SDSF enables authorized users to control job processing (hold, release, cancel, and purge jobs), monitor jobs, view the system log, edit JCL, and much more. Figure 2-2 on page 16 shows the main SDSF panel. From the main panel, press the F1 key to access SDSF help.



Figure 2-2 SDSF main panel

Within SDSF, the *Status Panel (ST)* (Figure 2-3) allows authorized users to display information about jobs, started tasks, and TSO users on the JES2 queues. The status panel is invoked via the **ST** command from the main SDSF panel. Summaries of some useful commands follow.

At the command prompt:

- ▶ **owner xyz**: Only display the status of jobs belonging to the specified user.
- ▶ **prefix xyz***: Only display the status of jobs that have the specified job name.
- ▶ **log**: Display the system log.

At the entry field next to the jobs:

- ▶ **s**: Display the job log produced by JES.
- ▶ **?**: Display more detailed job information (type **v** to view the details of items).

Session A - [32 x 80]

File Edit View Communication Actions Window Help

Display Filter View Print Options Help

SDSF STATUS DISPLAY ALL CLASSES

COMMAND INPUT ==>

NP	JOBNAME	JobID	Owner	PrtY	Queue	C	Pos	CHARS	'VSAM'	FOUND	ASys	Status	PAGE
?	VSAMTEST	JOB02975	ALEX03	1	PRINT		250						
	SMFCLR	STC02987	IBMUSER	1	PRINT		250						
	BPXAS	STC02973	IBMUSER	1	PRINT		260						
	BPXAS	STC02977	IBMUSER	1	PRINT		261						
	BPXAS	STC02982	IBMUSER	1	PRINT		262						
	BPXAS	STC02988	IBMUSER	1	PRINT		263						
	BPXAS	STC02978	IBMUSER	1	PRINT		264						
	BPXAS	STC02979	IBMUSER	1	PRINT		265						
	BPXAS	STC02974	IBMUSER	1	PRINT		266						
	BPXAS	STC02976	IBMUSER	1	PRINT		267						
	BPXAS	STC02931	IBMUSER	1	PRINT		268						
	BPXAS	STC02987	IBMUSER	1	PRINT		269						
	BPXAS	STC02994	IBMUSER	1	PRINT		270						
	BPXAS	STC02991	IBMUSER	1	PRINT		271						
	SMFCLR	STC02996	IBMUSER	1	PRINT		272						
	BPXAS	STC02988	IBMUSER	1	PRINT		273						
	BPXAS	STC02987	IBMUSER	1	PRINT		274						
	BPXAS	STC02990	IBMUSER	1	PRINT		275						
	BPXAS	STC02989	IBMUSER	1	PRINT		276						
	BPXAS	STC02995	IBMUSER	1	PRINT		277						
	BPXAS	STC02956	IBMUSER	1	PRINT		278						
	BPXAS	STC02993	IBMUSER	1	PRINT		279						
	BPXAS	STC02992	IBMUSER	1	PRINT		280						
	RC76	TSU02983	RC76	1	PRINT		281						

F1=HELP F2=SPLIT F3=END F4=RETURN F5=IFIND F6=BOOK
F7=UP F8=DOWN F9=swap nex F10=LEFT F11=RIGHT F12=RETRIEVE

Connected to remote server/host wtsc76.itso.ibm.com

Figure 2-3 SDSF status panel

The *Processes Panel (PS)* (Figure 2-4) allows authorized users to display information about z/OS UNIX System Service processes. It is invoked via the **ps** command from the main SDSF panel.

Session A - [32 x 80]

File Edit View Communication Actions Window Help

Display Filter View Print Options Help

SDSF PROCESS DISPLAY SC76 ALL LINE 1-7 (7)

COMMAND INPUT ==>

NP	JOBNAME	ID	PPID	ASID	ASIDX	LatchWaitPID	Command
	ALEX02	57	16777226	48	0030		rlogind2 -a -m xterm/38400
	ALEX02	58	57	48	0030		sh -L
	ALEX02	70	67108917	60	003C		sh -L
	ALEX02	53	16777226	52	0034		rlogind2 -a -m xterm/38400
	ALEX02	54	33554453	52	0034		sh -L
	ALEX02	04	16777270	60	003C		java Main
	ALEX02	17	16777226	60	003C		rlogind2 -a -m xterm/38400

F1=HELP F2=SPLIT F3=END F4=RETURN F5=IFIND F6=BOOK
F7=UP F8=DOWN F9=swap nex F10=LEFT F11=RIGHT F12=RETRIEVE

Connected to remote server/host wtsc76.itso.ibm.com

Figure 2-4 SDSF process panel

2.2.2 Writing JCL for a Java batch job

Just like any other program written for the mainframe operating environments, Java batch jobs must be submitted and controlled using JCL scripts. IBM provides a utility program called BPXBATCH to run shell scripts and executable files that reside in the HFS from JCL scripts. Sample JCL to invoke the simple Java program we discussed in the previous section is in Example 2-3.

Example 2-3 Sample JCL to invoke a Java program

```
//JAVAP JOB (ITS0),'USER1',REGION=30M,
//      CLASS=A,MSGCLASS=X,NOTIFY=&SYSUID
//* -----
/* Run Java program
/* -----
//RUN      EXEC PGM=BPXBATCH,
//          PARM='SH java -cp /u/user1/examples Example'
//SYSPRINT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//STDOUT   DD PATH='/u/user1/java.stdout',
//          PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//          PATHMODE=SIRWXU
//STDERR   DD PATH='/u/user1/java.stderr',
//          PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//          PATHMODE=SIRWXU
//STDENV   DD DUMMY
/* -----
/* Copy the output of Java program
/* -----
//COPYOUT  EXEC PGM=IKJEFT01,DYNAMNBR=300,COND=EVEN
//SYSTSPRT DD SYSOUT=*
//HFSOUT   DD PATH='/u/user1/java.stdout'
//HFSERR   DD PATH='/u/user1/java.stderr'
//STDOUTL  DD SYSOUT=*,DCB=(RECFM=VB,LRECL=133,BLKSIZE=137)
//STDERRL  DD SYSOUT=*,DCB=(RECFM=VB,LRECL=133,BLKSIZE=137)
//SYSPRINT DD SYSOUT=*
//SYSTSIN  DD DATA,DLM='>'
//          ocopy indd(HFSOUT) outdd(STDOUTL)
//          ocopy indd(HFSERR) outdd(STDERRL)
//          />
```

BPXBATCH takes either SH or PGM as a first parameter. When SH is specified, it expects the name of a shell script to run. When PGM is specified, it expects the name of an executable file in the HFS. The SH option is the default.

BPXBATCH uses four DD cards: STDOUT, STDIN, STDERR and STDENV. BPXBATCH directs the standard output of the step to STDOUT, the standard error to STDERR, the standard input from STDIN, and sets the USS

environmental variables of the step from STDENV. All of the DD cards are optional and have the default value of /dev/null. So, if any of the DD cards is not specified, it is simply ignored by the job.

When STDENV does not set the HOME and LOGNAME variables, they are set using the logon RACF® profile. When the SH option is used, variables set by STDENV are overridden by the values specified in /etc/profile, which are overridden again by the setting in the .profile file in the user's home directory.

The output of the standard output and error cannot be directly directed to the SYSPRINT output stream of JCL. Alternatively, in the JCL script, these output streams should be directed to temporary files (/u/user1/java.stdout and /u/user1/java.stderr, respectively). Then, the contents of these temporary files are copied back to the SYSPRINT output stream in the following job step (called COPYOUT). Consequently, the output of the Java program cannot be examined until the entire job step is completed. This is a known limitation of the BPXBATCH utility program.

Tip: The contents of temporary files used to store the standard output and error of Java program in HFS can be examined under USS using tools like **cat**, **head**, and **tail** while the batch Java program is running (**tail -f** displays the last several lines of the file and updates as more lines become available).

In the previous example, the environment variables are set using the profile file of the user who submitted the job before running the specified Java program. If different settings are required for a job, it is possible to set up the variables in one of the following three ways:

- Set the variables in JCL.

```
//STDENV DD *  
PATH=/bin  
JAVA_HOME=/usr/lpp/java/J1.4  
/*
```

- Set the variables in a file in HFS. It must be a text file with read-only access. The variables should be set as *variable=value* pairs in the file.

```
//STDENV DD PATH='/u/user1/env.profile',PATHOPTS=ORDONLY
```

- Set the variables in a data set. It can be a sequential, PDS, or SYSIN data set. The variables should be set as *variable=value* pairs in the data set.

```
//STDENV DD DSN=USER1.ENV.PROFILE',DISP=SHR
```

BPXBATCH is not limited to running batch Java programs; it is able to run any shell scripts or executable files residing in HFS.

2.2.3 Limitations of BPXBATCH

As we discussed in the previous section, BPXBATCH has the limitation of handling standard I/O streams when used within a JCL script. That is, the standard I/O streams must be directed to temporary files in HFS and copied back to a data set after the job step is completed.

Another limitation of BPXBATCH is its inability to allow executing programs to access DD cards defined in the enclosing JCL script. Traditionally, I/O requirements are described in a JCL script and made available to the running program as symbolic links in the mainframe environments. Hence, programs do not require re-compilation when different resources need to be used for I/O.

Example 2-4 illustrates the use of a DD card in a JCL script. The script invokes a REXX exec (called DDEXAMPL in Example 2-5) which reads the specified data set and simply prints out the contents of the specified resource. The DD card, INFILE, is used to specify a sequential data set to be processed by the REXX exec. By changing the value of INFILE DD card in the JCL script, it is possible to read a different data set, depending on the requirements, without re-writing the REXX exec.

Example 2-4 DD card example in JCL

```
//REXXJOB JOB (ITS0),'USER1',REGION=30M,
//          CLASS=A,MSGCLASS=X,NOTIFY=&SYSUID
//MSG      EXEC PGM=IRXJCL,PARM='DDEXAMPL'
//SYSTSIN DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//INFILE   DD DSN=USER1.DATA.TESTDAT,DISP=SHR
//SYSEXEC DD DSN=USER1.EXECS,DISP=SHR
```

Example 2-5 REXX exec utilizing DD card

```
/* REXX */

/* Read the specified data set in DD card called INFILE */
'EXECIO * DISKR INFILE 1 (FINIS STEM INLINES.)'

DO i = 1 TO INLINES.0
  /* Print each line. */
  SAY INLINES.i
END
```

This limitation is the consequence of the BPXBATCH utility program running the specified shell scripts or executable file in a separate address space. Hence, the values of DD cards are not visible by the invoked programs.

2.2.4 BPXBATSL

IBM provides another utility program called BPXBATSL to address the limitations of BPXBATCH. BPXBATSL provides an alternate entry point into BPXBATCH and forces a program to run using *local spawn* instead of *fork/exec*. Hence, all DD cards specified in a JCL script are made available in the invoked program.

Example 2-6 Example of BPXBATSL utility program

```
//RUNJAVA JOB (ITS0),'USER1',REGION=30M,
//          CLASS=A,MSGCLASS=X,NOTIFY=&SYSUID
//* -----
//* Run Java program
//* -----
//RUN      EXEC PGM=BPXBATSL,
//          PARM='PGM /usr/lpp/java/J1.4/bin/java Example'
//SYSPRINT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//STDOUT   DD PATH='/u/user1/java.stdout',
//          PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//          PATHMODE=SIRWXU
//STDERR    DD PATH='/u/user1/java.stderr',
//          PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//          PATHMODE=SIRWXU
//STDENV    DD PATH='/u/user1/env.profile',PATHOPTS=ORDONLY
//* -----
//* Copy the output of Java program
//* -----
//COMPOUT   EXEC PGM=IKJEFT01,DYNAMNBR=300,COND=EVEN
//SYSTSPRT  DD SYSOUT=*
//HFSOUT    DD PATH='/u/user1/java.stdout'
//HFSERR     DD PATH='/u/user1/java.stderr'
//STDOUTL   DD SYSOUT=*,DCB=(RECFM=VB,LRECL=133,BLKSIZE=137)
//STDERRL   DD SYSOUT=*,DCB=(RECFM=VB,LRECL=133,BLKSIZE=137)
//SYSPRINT  DD SYSOUT=*
//SYSTSIN   DD DATA,DLM='>'
//          ocopy indd(HFSOUT) outdd(STDOUTL)
//          ocopy indd(HFSERR) outdd(STDERRL)
//          />
```

Example 2-6 shows a sample JCL which uses BPXBATSL to invoke the sample Java program (Example). It is important to note that the BPXBATSL utility program does not set up the environmental variables using `/etc/profile` and the user's `.profile` file in the home directory. Instead, it relies solely on the environmental setting parameters specified in the JCL script. In this example, environmental variables are set in an HFS file called `/u/user1/env.profile`. Example 2-7 shows the contents of the file. In this file the `CLASSPATH`, `JAVA_HOME` and `PATH` variables are set.

Example 2-7 Environmental variable settings for BPXBATSL

```
CLASSPATH=./u/user1/examples
JAVA_HOME=/usr/lpp/java/J1.4
PATH=/usr/lpp/java/J1.4/bin
```

The usage of DD cards in Java batch jobs is discussed in the next section.

Tip: The full path for the Java program must be specified as an argument for the BPXBATSL utility program. If the name of the program becomes too long, then we recommend using symbolic links so that the correct name can be specified in the JCL script.

2.2.5 Comparison of JCLBATCH and JCLBATSL

Table 2-1 summarizes the characteristics of BPXBATCH and BPXBATSL.

Table 2-1 Comparison of BPXBATCH and BPXBATSL

	BPXBATCH	BPXBATSL
Inherit user profile	Yes	No
Return code	Returns the value returned from the running program.	Returns the value returned from the running program multiplied by 256. This is useful in determining errors in BPXBATSL invocation versus program errors in the running program.
DD cards supported	No	Yes
Use z/OS data sets	Yes (not via DD cards)	Yes
JVM runs in the same address space	No	Yes

2.3 Developing multi-step jobs

In this section, we discuss how to create a JCL script that has multiple job steps, and manage the execution flow of the job using conditional statements on each step and if-else statements enclosing multiple job steps.

2.3.1 Specifying multiple steps in a JCL script

One of the key strengths of a JCL script is its ability to control and manage multiple steps within a job and submit the job to the system for the execution. Depending on the availability of the system resources (such as CPU, memory, data set access) and the priority of the job, the operating system schedules the job. In this section, we use a simple scenario to describe the steps required to develop a multi-steps job and discuss the integration of legacy applications with a new Java application. Consider the following scenario:

There is a legacy application that produces a summary of transactions of customer accounts which occurred in a particular business day (called the transaction summary). There is another legacy application that updates the interest rate associated with each account given a data set that lists the account IDs and corresponding interest rates (called interest rate adjustment). Both applications are implemented using REXX execs. The company would like to develop a new program that uses the output of the transaction summary application to identify customers who use the system frequently so that the company can offer incentives to those customers (that is, adjust to a better interest rate). In the first stage of this project, the new program should produce an output that is suitable for the interest rate adjustment program. The logistics for determining candidate customers for the promotion should be implemented in a new program. In this scenario, the company decided to implement the new program in Java and integrate it with the legacy applications.

Data set I/O requirements for the legacy applications are specified in a JCL script and they are accessed in the legacy applications via corresponding symbolic names (no hard-coded data set names in the applications). For the transaction summary application, the results are written to a data set which is specified in the JCL. Similarly, the input for the interest rate adjustment program expects the input data set to be specified via DD cards in a JCL script¹. Provided all data set I/O requirements are specified in the JCL script, it is desirable to specify the data set I/O requirements for the new Java program in the JCL script as well. So, we use the BPXBATSL utility program to run the new Java program.

Example 2-8 shows a sample JCL script that integrates the legacy applications and the new Java application. The first step in the script (GENREC) invokes the transaction summary program called GENRCDS, which produces a summary of account transactions on a particular day. The output of the program is written to a data set called USER1.DATA(TRANRCDS), which is specified using the DD statement (OUTFILE). The second step of the script (JAVARUN) invokes the new Java application called ProcessTrans. Notice that the output of the first step

¹ It is possible to allocate DD cards dynamically in TSO environment using ALLOCATE rather than specifying everything in a JCL script. However we prefer to set everything in a JCL script for a job for the manageability of the job.

is specified as an input to the Java application using the DD statement (INTRANS). The value of the DD statement is passed to the Java application as the first command line argument (DD:INTRANS). The JCL script also specifies the name of an output data set (USER1.DATA(JAVAOUT)) using the DD statement (OUTDATA). This data set is used as an input for the last step of the script, which invokes the interest rate adjustment program.

Example 2-8 Using JCL to integrate legacy applications with a new application

```
//CUSTUDT JOB (ITS0),'USER1',REGION=30M,
//          CLASS=A,MSGCLASS=X,NOTIFY=&SYSUID
/** REXX EXEC TO PRODUCE THE RANDOM RECORD OF TRANSACTION.
/** IT PRODUCES A LIST OF CUSTOMER ID AND AMOUNT PAIRS.
/** IF THE AMOUNT IS NEGATIVE, IT IS A WITHDRAW TRANSACTION.
//GENREC   EXEC PGM=IRXJCL,PARM='GENRCDS'
//SYSTSIN  DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//OUTFILE  DD DSN=USER1.DATA(TRANRCDS),DISP=SHR
//SYSEXEC  DD DSN=USER1.EXECs,DISP=SHR
/** -----
/** RUN JAVA PROGRAM TO PROCESS THE TRANSACTION RECORD.
/** -----
//JAVARUN  EXEC PGM=BPXBATSL,
//          PARM='PGM /usr/lpp/java/J1.4/bin/java
//          -cp /u/user1 ProcessTrans DD:INTRANS'
//SYSPRINT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//STDOUT   DD PATH='/u/user1/java.stdout',
//          PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//          PATHMODE=SIRWXU
//STDERR   DD PATH='/u/user1/java.stderr',
//          PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//          PATHMODE=SIRWXU
//STDENV   DD DUMMY
//INTRANS  DD DSN=USER1.DATA(TRANRCDS),DISP=SHR
//OUTDATA  DD DSN=USER1.DATA(JAVAOUT),DISP=SHR
/** -----
/** Copy the output of Java program
/** -----
//COMPOUT  EXEC PGM=IKJEFT01,DYNAMNBR=300,COND=EVEN
//SYSTSPRT DD SYSOUT=*
//HFSOUT   DD PATH='/u/user1/java.stdout'
//HFSERR   DD PATH='/u/user1/java.stderr'
//STDOUTL  DD SYSOUT=*,DCB=(RECFM=VB,LRECL=133,BLKSIZE=137)
//STDERRL  DD SYSOUT=*,DCB=(RECFM=VB,LRECL=133,BLKSIZE=137)
//SYSPRINT DD SYSOUT=*
//SYSTSIN  DD DATA,DLM='>'
          ocopy indd(HFSOUT) outdd(STDOUTL)
          ocopy indd(HFSERR) outdd(STDERRL)
```

```

/>
/* REXX EXEC TO UPDATE THE INTEREST RATES OF CUSTOMERS.
//UPDATE EXEC PGM=IRXJCL,PARM='UPDATEIN'
//SYSTSIN DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//INDATA DD DSN=USER1.DATA(JAVAOUT),DISP=SHR
//CUSTDATA DD DSN=USER1.DATA(CUSTINT),DISP=SHR
//SYSEXEC DD DSN=USER1.EXECS,DISP=SHR

```

Section 3.1, “The JRIO library” on page 32 describes the Java Record I/O (JRIO) library to access the data set and HFS on zSeries, as well as how to access the resources specified using the DD statements in JCL scripts.

2.3.2 Conditions on a step

The EXEC statements in a JCL script have several parameters that enable system programmers to specify under what conditions the step should be executed. Several parameters that are useful in controlling the execution of steps are discussed in this section.

COND	Checks the return codes of previous steps to determine if this step should be executed or bypassed.
MEMLIMIT	Specifies the limit on the total number of usable virtual pages above the bar in a single address space.
REGION	Specifies the amount of space required by the step.
TIME	Specifies the maximum time the step is allowed to use the processor.

Example 2-9 shows how to specify various conditions under which the step should be executed. In this example, the step RUN2 is executed only when the previous step, RUN1 is completed successfully (that is, a return code of 0 is received from the step). Furthermore, only 10 megabytes of space can be used to execute the step.

Example 2-9 Specifying condition on a job step

```

//RUN2 EXEC PGM=BPXBATCH,COND=(0,NE,RUN1),REGION=10M,
// PARM='SH java Main'
//SYSPRINT DD SYSOUT=*
//SYSOUT DD SYSOUT=*
//STDOUT DD *
//STDERR DD *
//STDENV DD *

```

2.3.3 Conditional step execution using IF-ELSE

Rather than specifying conditions individually on a job step, it is possible to enclose multiple steps within an IF-ELSE statement in a JCL script. Conditional expressions for an IF-ELSE statement can be constructed using comparison operators including GT, LT, NG, NL, EQ, NE, GE, and LE, and logical operators (AND and OR). Also RC, ABEND, and RUN keywords are provided to assess the status of steps in a job.

RC	Holds the value of the greatest return code in the previous steps in a job. RC.STEPNAME allows the retrieval of the return code from a particular step in a job.
ABEND	Indicates whether any job step has abended in the job. ABEND.STEPNAME indicates whether a particular step in a job has abended in the job.
RUN	Indicates if steps in a job have run. It is possible to determine if a particular step has run using RUN.STEPNAME.

Example 2-10 illustrates the use of the IF-ELSE statement in a JCL script. In this example, the job step RUNIF gets executed if the greatest return code of the previous steps is zero. If this condition is true, the Java program called SendResult is invoked. If the greatest return code is not zero, then it will execute steps called ELSE1 and ELSE2. In those steps, cleanup processes are invoked to handle the situation where previous job steps have failed.

Example 2-10 Example of IF-ELSE statement in JCL script

```
// IF (RC = 0) THEN
//RUNIF EXEC PGM=BPXBATCH,
//      PARM='SH java
//      -cp /u/user1/examples SendResult'
//SYSPRINT DD SYSOUT=*
//SYSOUT DD SYSOUT=*
// ELSE
//ELSE1 EXEC PGM=BPXBATCH,
//      PARM='SH java
//      -cp /u/user1/examples CleanUpTmpData'
//SYSPRINT DD SYSOUT=*
//SYSOUT DD SYSOUT=*
//ELSE2 EXEC PGM=BPXBATCH,
//      PARM='SH java
//      -cp /u/user1/examples NotifyFailure'
//SYSPRINT DD SYSOUT=*
//SYSOUT DD SYSOUT=*
// ENDIF
```

2.4 Design considerations for multi-step jobs

It is possible to invoke multiple Java programs from a shell script that resides in the HFS, rather than invoking Java programs individually from JCL scripts. Consider the factors discussed in this section when making the design decision.

2.4.1 What skill sets are available in the development team?

Depending on the skill sets of the development team, the team needs to find a solution that balances the differences between z/OS-oriented programmers and UNIX-oriented programmers. The right solution is one in which the work done in each world is transparent to the other world, so that everybody has a good level of understanding of the overall process implemented by the job. Such a solution would facilitate the communication between the team members and easier problem determination.

2.4.2 How much integration with legacy applications is necessary?

The expected purpose of a new Java application may help determine how it should be integrated within a job. On one extreme you can have a situation where you are developing a brand new application with minimal integration required with legacy applications. In such cases, it is entirely up to the Java application development team to decide what would be the best way to deliver the solution.

On the other extreme you can have a situation where the new application is expected to replace the functionality of a legacy application in the company's effort to modernize their system. In such cases, the surrounding legacy systems may impose rigid requirements on the new Java application (for example, there is an error recovery process that must be invoked when the new Java application fails). The company has no plan to rewrite the error recovery system until the management team has seen the value of the legacy modernization effort. In such cases, the Java development team must develop the new application with a set of strict requirements for input and output of the new application that will be controlled by the legacy system.

2.4.3 How much control is required between steps?

Because UNIX shell scripting is a powerful scripting language in its own right, it might be more desirable to describe the steps required for a particular job in a shell script rather than in a JCL script.

Example 2-11 shows a simple UNIX shell script that invokes a Java application and performs different actions depending on the completion status of the Java application.

Example 2-11 Simple example of UNIX shell script

```
#!/bin/sh

java -cp $HOME/examples Main
rc=$?
if ( test $rc -ne 0 ) then
    # Case when the Java program failed.
    # Error recovery should be inserted here.
    echo "Java program failed. RC =" $rc
else
    # Case when the Java program successfully finish.
    echo "Java program completed with success!"
fi

exit $rc
```

2.4.4 Return code from BPXBATCH and BPXBATSL

BPXBATCH and BPXBATSL have slightly different behaviors in terms of how they return the return code from the running program. BPXBATCH simply returns the value returned from the running shell or program. BPXBATSL returns the value returned from the running shell or program multiplied by 256. This is to distinguish between the value returned by the running shell or program and the return code from the BPXBATSL itself.

Careful consideration is required when dealing with the return code of a step involving BPXBATCH and BPXBATSL. For example, BPXBATCH and BPXBATSL return the same return code² when the Java runtime environment could not find the class specified *and* an exception was thrown by the Java program and terminated. In both cases, the examination of output from the standard error stream is required to determine the problem.

To facilitate easier problem determination, we recommend designing a set of return codes for a new Java application at the beginning of the project, and requiring the Java development team to adhere to these standards. A **System.exit(rc)** call should be inserted to properly return the status of the Java program to the job. Example 2-12 illustrates how to return the status of a Java application properly to the job. When the application finishes without error, then it should exit with the return code of zero. When an exception is thrown in the program, it calls a lookup function (**rcLookup(...)**) to exit the program with the

² 1 for BPXBATCH and 10 for BPXBATSL.

agreed return code. The lookup function should return the appropriate return code, which describes the status of the Java program so that a system programmer writing a JCL script can perform appropriate action in the enclosing JCL script.

Example 2-12 Simple example of how to exit properly from a Java application

```
try {  
    // Do something.  
    ...  
    System.exit(0);  
} catch (Exception e) {  
    System.exit(rcLookup(e));  
}
```

Access to MVS data sets

For Java, without specifically talking about z/OS, the most important data stores are files and relational databases. For access to relational databases (and even non-relational ones), the Java DataBase Connectivity (JDBC) APIs can be used in both an application server and a stand-alone environment. Libraries with classes and Beans are available for access to files.

In a z/OS environment, however, there is a third important issue: access to MVS™ data sets. In the UNIX environment of z/OS one could perfectly live with the generic file access classes of Java to access files stored in the UNIX format in the HFS. In the z/OS environment, however, the usual way to store data is in MVS data sets. In this chapter we discuss how these MVS data sets can be accessed. We do not address JDBC or generic file access from Java because these topics are documented in numerous other places already.

3.1 The JRIO library

The Java Record I/O (JRIO) library lets Java applications access traditional mainframe file systems which the Java I/O library does not support. In addition, it provides the *record-oriented view* of the data stored in a file and provides facilities to access the records sequentially, randomly, or using keys. JRIO is an integral part of all SDKs available for the z/OS platform. Refer to the following Web site for the product information and the instructions for download and installation:

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

The JRIO library enables Java applications to access the following types of mainframe file systems:

- ▶ Partitioned Data Set (PDS) and Partitioned Data Set Extended (PDSE)
- ▶ Sequential files
- ▶ Virtual Sequential Access Method (VSAM) data sets (only KSDS)
- ▶ Hierarchical File System (HFS)

In the following sections, we present an overview of the JRIO library. We provide examples to access records in data sets either sequentially or randomly in this section.

3.1.1 JRIO classes and interfaces

High-level qualifiers of data sets and directories in a Hierarchical File System are represented using the *IDirectory* interface and its concrete implementation class called *Directory*. The interface defines methods for creation and deletion as well as methods for querying the status of the underlying object such as name, read/write permissions, existence, primary and secondary allocation, and dates. An instance of the concrete implementation of the interface should be instantiated as follows:

```
IDirectory dir = Directory.getInstanceOf("//USER1.DATA");
```

A member in a data set or file in the HFS can be represented using the *IRecordFile* interface and its concrete implementation class called *RecordFile*. The interface defines methods for creation and deletion as well as methods for querying the status of the underlying object similar to the *IDirectory* interface. An instance of the concrete implementation of the interface should be instantiated as follows:

```
IRecordFile file = RecordFile.getInstanceOf("//USER1.DATA(DATA1)");
```

Tip: When accessing a data set, the name of the data set must be preceded with // to distinguish the data set from the directories and files residing in HFS.

The following example refers to a member called DATA1 in a data set called USER1.DATA.

```
//USER1.DATA(DATA1)
```

The following example refers a file called MyJavaClass.java in a directory called /u/user1 in HFS.

```
/u/user1/MyJavaClass.java
```

When a data set name is passed to a Java program via a JCL DD card, the name should have the format of DD:INDATA where INDATA is the DD card name used in the calling JCL script. In the Java program, the DD card name should be accessed as follows:

```
IRecordFile file = RecordFile.getInstanceOf("//DD:INDATA");
```

The JRIO library provides a record-oriented view of the data stored in a file. The data can be accessed sequentially, randomly, or using keys. To enable record-oriented access of the data, the JRIO provides the Java interfaces and corresponding concrete implementation classes that are summarized in Table 3-1.

Table 3-1 Access methods and corresponding JRIO classes

Access method	JRIO classes to use
Sequential	IFileInputRecordStream IFileOutputRecordStream
Random	IRandomAccessRecordFile
Key	IKeyedAccessRecordFile

IFileInputRecordStream and IFileOutputRecordStream define methods for accessing records in a file sequentially. The IFileInputRecordStream interface defines methods for reading records from a file. It provides several read methods for different purposes. Using these methods, it is possible to read one record at a time. The concrete implementation of the interface should be instantiated as follows:

```
IFileInputRecordStream input =  
FileInputRecordStream.getInstanceOf(file);
```

The `IFileOutputRecordStream` interface defines methods for writing records to a file. It provides several write methods that can be used to write one record at a time. The concrete implementation of the interface can be instantiated as follows:

```
IFileOutputRecordStream out =  
FileOutputRecordStream.getInstanceOf(file);
```

The `IRandomAccessRecord` defines methods required for accessing records in a file randomly. Some of the methods defined in the interface are the following:

- | | |
|-----------------------------|---|
| positionFirst() | Positions the cursor at the beginning of the records in a file. The first record is ready for reading/writing. |
| positionLast() | Positions the cursor at the end of the records in a file. The last record is ready for reading/writing. |
| positionNext() | Moves the cursor to the next record in a file. |
| positionPrevious() | Moves the cursor to the previous record in a file. |
| seek(int recordNo) | Moves the cursor forward for the specified number of records in a file. |
| read(byte[] buffer) | Reads a record from the current cursor location. If the record length is greater than the specified buffer size, then trailing data of the record is discarded. If the record is shorter than the specified buffer, then only the record-length bytes of data will be read. |
| write(byte[] buffer) | Write the record-length bytes of data from the specified buffer to a file. If the length of the buffer is greater than the record-length of the file, then only the record length bytes of data will be written. If the record length of the data is longer than the buffer length, then additional bytes will be padded to the record. |

A concrete implementation of the interface should be instantiated using the following method:

```
IRandomAccessRecord ra = RandomAccessRecord(file, mode);
```

The first parameter specifies the file to be opened for random access. The second parameter specifies the accessing mode of the file (read-only access or read-write access).

The `IConstants` interface defines a set of constants that are useful when working with the JRIO library. Some of the commonly used constants are the following:

- I/O modes
 - JRIO_READ_MODE

- JRIO_READ_WRITE_MODE
- Data set characteristics
 - JRIO_DEFAULT_RECORD_FORMAT: Fixed block record.
 - JRIO_DEFAULT_RECORD_LENGTH: Default record length of 80.
 - JRIO_VARIABLE_MODE
 - JRIO_FIXED_MODE
 - JRIO_DEFAULT_DSP_TYPE: Same as JRIO_DSP_TYPE_OLD.
 - JRIO_DSP_TYPE_OLD
 - JRIO_DSP_TYPE_SHR
 - JRIO_DEFAULT_ALLOC_TYPE: Same as JRIO_BLOCK_ALLOC_TYPE.
 - JRIO_BLOCK_ALLOC_TYPE
 - JRIO_CYL_ALLOC_TYPE
 - JRIO_TRACK_ALLOC_TYPE

The classes and interfaces provided for accessing records using keys are described in 3.2, “Working with VSAM data sets” on page 41.

3.1.2 JRIO usage examples

In this section we present a number of examples using JRIO.

Copying members of a data set

The following Java program demonstrates the use of the JRIO library. The program takes an existing data set name, creates a new data set with the same characteristics, and copies all members of the specified data set into the new data set. This example demonstrates how to use IDirectory and IRecordFile to represent a data set and members in the data set respectively. Using the instances of IDirectory and IRecordFile, we also demonstrate how to discover all the members in a data set, create new members in a data set with the same characteristics, and copy all members into the new data set. We use IFileInputRecordStream and IFileOutputRecordStream to read records from a member and write the records to a new member.

The program can be invoked as follows:

```
java DirCopy USER1.DATA
```

The first argument to the program specifies the data set name whose members are copied to a new data set. The new data set is created with the name USER1.DATA.COPY. When the program executes successfully, it prints the

names of members being copied from and the corresponding new member names in the new data set.

The program checks whether the specified data set exists using the method called **checkSourceDir(...)**. In this method, an instance of the Directory class is created with the specified data set name and its existence is checked using the **exists()** method. If the specified data set does not exist, then it prints out an error message and quits.

If the specified data set exists, then it creates another instance of the Directory class to represent the new data set where all members of the specified data set are copied. It creates a new data set with the same characteristics as the specified data set using the **mkdirLike(...)** method. The method takes a reference to an existing data set, which is used as a guideline to create the new data set.

Once a new data set is created, it copies all members in the specified data set to the new data set being created by the program in the **copyDir(...)** method. The list of all members of the specified data set is retrieved using the **list(...)** method of the IDirectory interface. The method returns a list of members as an array of Strings. The program loops through the list of member names to copy one member at a time. In each iteration, it creates instances of IRecordFile which represent a member to be copied from and a member to be copied to. The destination member is created with the same characteristics as the original member using the **createFileLike(...)** method. Once the new member is created, it copies the contents of the member using the **copyFile(...)** method.

In the **copyFile(...)** method, it creates an instance of IFileInputRecordStream to read data from the specified member and an instance of IFileOutputRecordStream to write data to the new member in the new data set. It reads one record at a time from the original member and writes the record to the new file. The size of the record in the specified member is determined using the **getRecordLength()** method, which is used to allocate enough space to temporarily store the record in a byte array buffer. Once all records are copied to the new member, it closes the input stream from the original member, and flushes and closes the output stream to the new member.

Example 3-1 Java program that copies all members of a data set to another data set

```
import java.io.IOException;
import com.ibm.recordio.*;

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

    public static void main(String args[]) {
```

```

if (args.length != 1) {
    System.err.println("Usage: DirecotoryCopy dirName.");
    System.exit(8);
}

try {
    IDirectory srcDir;
    if ((srcDir = checkSourceDir("//" + args[0])) == null) {
        System.err.println("Source directory " + args[0] + " does not exist!");
        System.exit(8);
    }

    IDirectory copyDir = Directory.getInstanceOf("//" + args[0] + ".COPY");
    if (!copyDir.mkdirLike(srcDir)) {
        System.err.println("Failed to create copy directory: " + copyDir.getName());
        System.exit(8);
    }

    copyDir(srcDir, copyDir);

} catch (Exception e) {
    e.printStackTrace();
    System.exit(8);
}
}

// Checks if the specified directory exists.
private static IDirectory checkSourceDir(String dirName) {
    IDirectory dir = Directory.getInstanceOf(dirName);
    if (dir.exists()) {
        return dir;
    } else {
        return null;
    }
}

// Loops through the specified directory and copy all children.
private static void copyDir(IDirectory src, IDirectory dest) throws IOException {
    String[] children = src.list();
    for (int i = 0; i < children.length; i++) {
        IRecordFile srcFile = RecordFile.getInstanceOf(src.getPath() + "(" + children[i] + ")");
        IRecordFile destFile = RecordFile.getInstanceOf(dest.getPath()
            + "(" + children[i] + ")");
        if (!destFile.createFileLike(srcFile)) {
            System.err.println("Failed to copy " + srcFile.getPath() + ".");
        } else {
            copyFile(srcFile, destFile);
        }
    }
}

```

```

    }
}

// Copies the contents of src file to the destination file.
private static void copyFile(IRecordFile srcFile, IRecordFile destFile)
    throws IOException {
    byte[] buffer;
    buffer = new byte[srcFile.getRecordLength()];

    FileInputStreamRecordStream in = FileInputStreamRecordStream.getInstanceOf(srcFile);
    FileOutputStreamRecordStream out = FileOutputStreamRecordStream.getInstanceOf(destFile);

    while (in.read(buffer) != -1) {
        out.write(buffer);
    }

    in.close();
    out.flush();
    out.close();
    System.out.println("Copied from " + srcFile.getPath() + " to " + destFile.getPath());
}
}

```

Accessing records randomly

The following Java program demonstrates how to use `IRandomAccessRecordFile` to randomly access records within a file. This example demonstrates how to read the file sequentially and randomly using the class. The program can be invoked as follows:

```

java RandomAccess "USER1.DATA(MYDATA1)"
or
java RandomAccess USER1.DATA.SEQDATA1

```

The first invocation example reads data from a member of a PDSE data set. The second invocation example reads data from a sequential data set. The JRIO library hides the differences in the underlying objects represented (PDSE versus sequential file) and provides a consistent interface.

The program creates an instance of `IRecordFile` object using `getInstanceOf(...)` method. Using the reference to the `IRecordFile` object, it queries the existence of the specified file and readability using `exists()` and `canRead()` methods respectively. If either condition is false, then it prints out an error message and quits.

The program creates an instance of `IRandomAccessRecordFile` using the **`getInstanceOf(...)`** method. The second parameter of the method specifies how the file should be opened (open for read or read and write).

In **`readSequentially(...)`** method, all records in the specified file are read sequentially using the **`read(...)`** method. The **`read(...)`** method reads one record at a time, positions the cursor at the beginning of the next record, and returns the number of bytes read. It is important to provide a byte array buffer that is large enough to hold a record entirely. If the specified buffer is too small, then only the size of the buffer is read into the buffer and the remaining data is discarded. Likewise, if the specified buffer is too large, then it only reads record-length bytes at a time (the buffer must be initialized each time). To avoid problems with the buffer size, the buffer should be allocated using the size returned by the **`getRecordLength()`** method. The method returns the size of an individual record stored in a file. If variable length records exist in the specified file, then it returns the largest record length possible in the file.

In the **`readRandom(...)`** method, we demonstrate how to access records randomly in a file. To access records randomly, the cursor within a file needs to be positioned at the beginning of the record to be read next. To position the cursor, the `IRandomAccessRecordFile` defines several methods such as **`positionFirst()`**, **`positionLast()`**, and **`seek(...)`**. In this example, we demonstrate:

1. How to read the last record in a file.
2. How to position the cursor for the first record, skip one record and read the next record in a file.
3. How to position the cursor for the first record, skip two records and read the third record in a file.

Example 3-2 Java program which accesses records in a file randomly

```
import java.io.IOException;
import com.ibm.recordio.*;

public class RandomAccess {

    public static void main(String args[]) {
        if (args.length != 1) {
            System.err.println("Usage: RandomAccess name");
            System.exit(8);
        }

        try {
            IRecordFile file = RecordFile.getInstanceOf("//" + args[0]);
            if (!file.exists() || !file.canRead()) {
                System.err.println("Specified file does not exist or cannot read: " + args[0]);
            }
        }
    }
}
```

```

        System.exit(8);
    }

    IRandomAccessRecordFile randomFile
        = RandomAccessRecordFile.getInstanceOf(file, IConstants.JRIO_READ_MODE);

    readSequentially(randomFile);

    readRandom(randomFile);

} catch (Exception e) {
    e.printStackTrace();
    System.exit(8);
}
}

// Reads the randomly accessible file sequentially.
private static void readSequentially(IRandomAccessRecordFile randomFile)
    throws IOException {
    byte[] buffer = new byte[randomFile.getRecordLength()];

    // Position at the beginning of the file.
    randomFile.positionFirst();

    while (randomFile.read(buffer) != -1) {
        // Read bytes into buffer. Do something...
        System.out.println(new String(buffer));
    }
}

// Demonstrates usage of methods supporting random access.
private static void readRandom(IRandomAccessRecordFile randomFile)
    throws IOException {

    byte[] buffer = new byte[randomFile.getRecordLength()];

    // Position to the last record.
    randomFile.positionLast();
    randomFile.read(buffer);
    System.out.println(new String(buffer));

    // Position first, and skip first record.
    randomFile.positionFirst();
    randomFile.positionNext();
    randomFile.read(buffer);
    System.out.println(new String(buffer));

    // Position first, and skip 2 records.
    randomFile.positionFirst();

```

```
randomFile.seek(2);
randomFile.read(buffer);
System.out.println(new String(buffer));
}
}
```

Tip: It is possible to get detailed tracing of the JRIO library by invoking the application as follows:

```
java -DRIOJADEBUG MyApp 1> trace.out 2>&1
```

3.2 Working with VSAM data sets

In this section, we describe VSAM data sets including their characteristics and allocation process, and demonstrate how to access such data sets using the JRIO library with a simple example.

3.2.1 VSAM overview

IBM introduced Virtual Sequential Access Method (VSAM) in the early 1970s to facilitate sequential, random, and keyed access of records. There are four different types of VSAM data sets: Key-Sequenced Data Set (KSDS), Entry-Sequenced Data Set (ESDS), Relative Record Data Set (RRDS) and Linear Data Set (LDS). Many products are available to work with VSAM data sets, including CICS, File Manager, and DITTO, just to name a few. Several programming languages also provide mechanisms to support VSAM data sets, including COBOL, PL/1, Assembler, and Java. The JRIO library only provides support for KSDS VSAM data sets.

Designed by application developers, a *logical record* is a unit of information used to store data in a VSAM data set. For example, a record may contain customer ID, name, address and account number. An important field in the logical record is the *key*. Its contents can be used to retrieve the specific logical record. A set of related components which constitute a VSAM data set is called a *cluster*. For a KSDS, a cluster is the set of a data component and an index component. The concept of clusters simplifies VSAM processing, providing a way to treat index and data components as a single entity, with its own catalogued name.

VSAM data sets can be defined using a utility program called IDCAMS, which is included in DFSMSdfp. A sample JCL to create a VSAM file is in Example 3-3. In this example, a VSAM data set is created with a 10-byte key field and 28-byte fixed data size. After the allocation is completed, there should be a USER1.VSAM.CLUSTER (cluster), USER1.VSAM.CLUSTER.INDEX (index file)

and USER1.VSAM.CLUSTER.DATA (data file). From a Java application program, VSAM data sets are referred to by their cluster names. VSAM data sets must be allocated before Java applications can work with the data sets (no support is provided to allocate new VSAM data sets from Java applications in the JRIO library).

Example 3-3 JCL to create a VSAM file

```
//VSAM JOB (ITS0),'USER1',CLASS=A,
// NOTIFY=&SYSUID,MSGCLASS=T
/* -----
/* Create VSAM file.
/* -----
//VSAMALOC EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
    DEFINE CLUSTER (NAME('USER1.VSAM.CLUSTER') -
                    CYL(50 10) -
                    VOL(BH6ST2) -
                    KEYS(10 0) -
                    RECORDSIZE(38 38) -
                    SHAREOPTIONS(4,3) -
                    IMBED SPEED UNIQUE SPANNED) -
    INDEX (NAME('USER1.VSAM.CLUSTER.INDEX')) -
    DATA (NAME('USER1.VSAM.CLUSTER.DATA') -
          CISZ(2048) -
          FREESPACE(50 50))

/*
```

3.2.2 Accessing records in a VSAM data set using the JRIO library

The JRIO library provides a Java interface called `IKeyedAccessRecordFile` and its concrete implementation class called `KeyedAccessRecordFile` to manipulate records stored in a VSAM data set. An instance of `IKeyedAccessRecordFile` should be created as follows:

```
IKeyedAccessRecordFile karf =
KeyedAccessRecordFile.getInstanceOf(clusterName,
    IConstants.JRIO_READ_WRITE_MODE);
```

It returns a reference to an object which implements the `IKeyedAccessRecordFile`. The first parameter specifies the name of the cluster file of the VSAM data set. The second parameter specifies how the data set will be accessed, either read-only or read and write.

When working with a VSAM data set, most of the methods defined in the `IKeyedAccessRecordFile` require a reference to a key file of the VSAM data set. A reference to the key file of the VSAM data set can be accessed as follows:

```
IRecordFile keyFile = karf.getPrimaryIndex();
```

To demonstrate how to use the key file as an argument to a method, the following line of code reads the current record from the VSAM data set:

```
karf.positionForward(keyFile, key);  
karf.read(keyFile, buffer);]
```

In this example, the cursor in the key file is forwarded to the record which has the specified key. The read method reads the record that corresponds to the record pointed to by the cursor. After the read operation, the position of the cursor is forwarded to the next record.

Some of the most commonly used methods are the following:

positionFirst	Positions the cursor at the first record in the specified index file.
positionLast	Positions the cursor at the last record in the specified index file.
positionFoward	Moves the cursor to a record that matches the specified key, which is specified in the second parameter of the method.
read	Reads a record from the current position of the cursor.
write	Writes a record to the current position of the cursor.
update	Updates the record at the current position of the cursor. This method must be called immediately after calling the read method.
deleteRecord	Deletes the record at the current position of the cursor.

The **read(...)**, **write(...)** and **update(...)** methods take an array of bytes as an argument to perform their corresponding functionality. The size of an array should match the size of record which is read, written, or updated, including the key field and the data field.

Example 3-4 shows a sample Java program that manipulates records stored in a VSAM data set. The **readSequentially(...)** method shows how to read records in a VSAM data set sequentially. Each call to the **read(...)** method moves the cursor to the next record in the VSAM data set. When the last record in the data set is read, it returns -1.

The **updateRecord(...)** method updates the contents of a record that is specified by the key. It reads the record that corresponds to the specified key, then it manipulates the record and updates the record in the VSAM data set.

Example 3-4 Java program to manipulate KSDS VSAM data set

```
import java.io.IOException;
import com.ibm.recordio.*;

public class VSAMExample {

    public static void main(String args[]) {

        try {
            String clusterName = "//USER1.VSAM.CLUSTER";
            IRecordFile rf = RecordFile.getInstanceOf(vFile, 38, IConstants.JRIO_FIXED_MODE);
            IKeyedAccessRecordFile karf = KeyedAccessRecordFile.getInstanceOf(rf,
                IConstants.JRIO_READ_WRITE_MODE);

            // Get a reference to the VSAM key file.
            IRecordFile keyFile = karf.getPrimaryIndex();

            // Read data sequentially.
            readSequentially(karf, keyFile);

            // Create a key for a record.
            String myKey = "0000000004";
            IKey key = Key.getKey(myKey.getBytes());

            // Update the record.
            updateRecord(karf, keyFile, key);

            // Close the output stream for VSAM file.
            karf.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("All done!");
    }

    public static void readSequentially(IKeyedAccessRecordFile karf, IRecordFile keyFile)
        throws IOException {
        byte[] buffer = new byte[38];
        // Position the cursor to the beginning of the recrods.
        karf.positionFirst(keyFile);
        while (karf.read(keyFile, buffer) != -1) {
            // Construct the string and print.
            String s = new String(buffer);
        }
    }
}
```

```

        System.out.println "[" + s + ""];
    }
}

public static void updateRecord(IKeyedAccessRecordFile karf, IRecordFile keyFile, IKey key)
    throws IOException {
    byte[] buffer = new byte[38];

    // Position the cursor at the first record.
    karf.positionFirst(keyFile);

    // Position the cursor to the specified record.
    karf.positionForward(keyFile, key);

    // Read the current data for the specified key.
    System.out.println("Read " + karf.read(keyFile, buffer) + " bytes..");

    // Update the record in the buffer.
    byte[] data = "Some new data.....".getBytes();
    // Update only the data field.
    System.arraycopy(data, 0, buffer, 10, 28);

    karf.update(keyFile, buffer);
}

```

Tip: When working with data sets using the JRIO library, it is important to specify the characteristics of the data sets explicitly before using them in case the characteristics of the data sets are different from the default values. For example, when working with a VSAM file containing data whose record length is not equal to the default value (that is, 80), the record length must be explicitly specified before the data can be updated or written. If the characteristics of data sets are not properly specified, exceptions will be thrown. The following example demonstrates how to instantiate `IKeyedAccessRecordFile` for a VSAM file which has a variable record length:

```

IRecordFile rf = RecordFile.getInstanceOf(vFile, maxRecLen,
    IConstants.JRIO_VARIABLE_MODE);
IKeyedAccessRecordFile karf = new KeyedAccessRecordFile(rf,
    IConstants.JRIO_READ_WRITE_MODE);

```

In this example, `vFile` is the name of the VSAM data set to process, `maxRecLen` specifies the maximum record length of the data set (since it is variable record length data), and `IConstants.JRIO_VARIABLE_MODE` is specified to indicate it is a VSAM data set with the variable record length.

3.3 Working with COBOL packed decimal numbers

COBOL programs store numbers internally in several different ways. One of these internal representations is called *COBOL packed decimal numbers*. It stores numbers in a calculation-friendly format and uses the minimum storage depending on how many digits are required to store the number for an application. Packed decimal numbers occupy one byte of storage for every two decimal digits, except that the rightmost byte contains only one digit and the sign. That is, it uses 4 bits to represent either a digit or sign. A variable of such type is declared in a COBOL program as follows:

```
MYVAR1 PIC S999 USAGE COMP-3.  
MYVAR2 PIC 999  USAGE COMP-3.
```

The first variable, MYVAR1, is a signed COBOL packed decimal number that uses 3 bytes to store numbers (that is, up to five digits of signed numbers). The second variable, MYVAR2, is an unsigned COBOL packed decimal number that uses 3 bytes to store numbers (that is, up to five digits of unsigned numbers). Unlike many other numerical representations of programming languages, including Java, it uses the last 4 bits to represent sign. Therefore the first variable is able to store values from -99999 to 99999, and the second variable is able to store values from 0 to 99999.

When such values are read from a data set using the JRIO library, they are returned as an array of bytes. It is the responsibility of the application developer to interpret the byte arrays appropriately and convert them into suitable Java types. Figure 3-1 shows the internal representations of a signed number 23 as a COBOL packed decimal and a Java integer. For the simplicity of the discussion, we assume this number is stored in a 2-byte field¹. In a 2-byte field, it is possible to store numbers with 3 digits. The first byte field contains a hex decimal value of 02, which indicates the first two digits of this number are 0 and 2. The second byte field has a hex decimal value of 3C, which indicates the last digit of this number is 3, and it is a positive number. The last four bits of the field is used to represent the sign of the number stored.

Using 4 bits, it is possible to represent 16 different values. The use of different bit combinations for COBOL packed decimal numbers is summarized in Table 3-2. 0 to 9 are used to represent a digit, C indicates a positive number, D indicates a negative number, F indicates an unsigned number, and the rest are not used. C, D and F can only appear in the last 4 bits of a field.

In this section we describe an approach to convert such numbers to and from Java integers. JRecord APIs discussed in 3.4, “JRecord bean generator and

¹ It is possible that this number is stored in a variable that is declared to occupy more bytes. In such cases, 0s are padded at the beginning of unused bytes to represent a number. For example, a 4-byte field would have a value of 0x0000023C for +23.

supporting APIs” on page 50 provide a set of utility methods to perform such conversions.

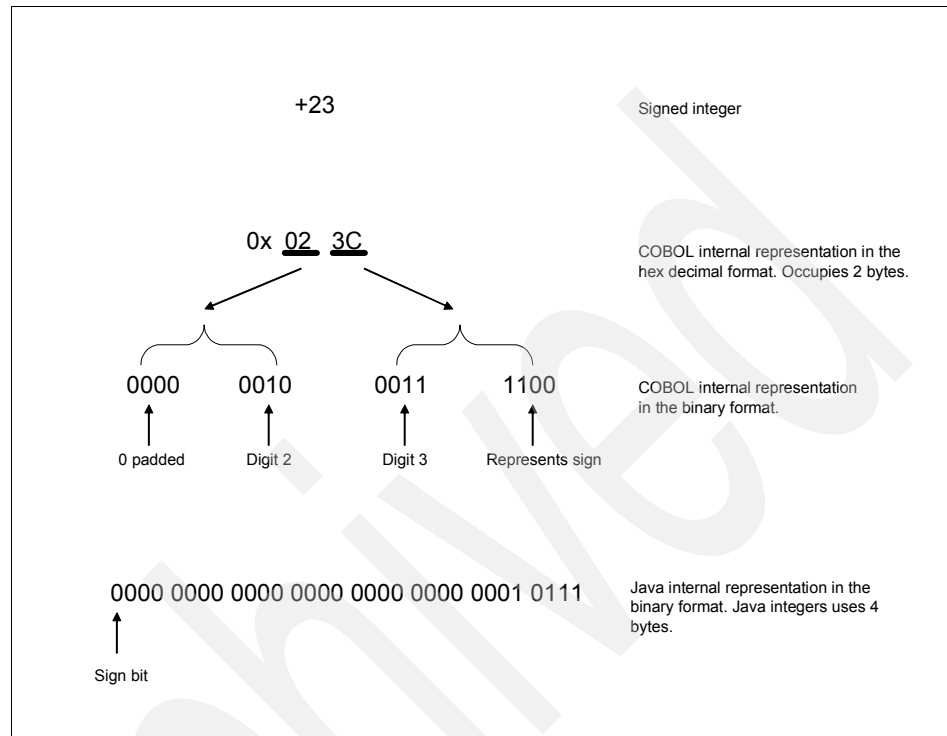


Figure 3-1 Conversion from COBOL packed decimal to Java integer

Table 3-2 Summary of 4-bits usage with COBOL packed decimals

Values in hex decimal	Meaning
0 ~ 9	Digits
C	Positive number
D	Negative number
F	Unsigned number
A, B, and E	Not used

3.3.1 Converting from COBOL packed decimals to Java integers

In this section, we describe a technique to convert an array of bytes containing a number in the COBOL packed decimal format to an equivalent value in the Java integer.

As we discussed in the previous section, packed decimal numbers occupy one byte of storage for every two decimal digits, except that the rightmost byte contains only one digit and the sign. Consequently, the upper and lower 4 bits must be examined separately to interpret the number correctly. The following steps summarize the procedure required for the conversion:

1. Determine the number of digits.
2. Determine the multiplier for the most significant digit (number of digits multiplied by 10).
3. Initialize an integer total value to zero (this field will be the result of the conversion process).
4. For each byte from the most significant byte to the least significant byte, perform the following steps:
 - a. Retrieve two digits by examining upper and lower 4 bits.
 - b. For the upper digit, multiply by the multiplier and add the result to the total. Divide the multiplier by 10 for the next digit.
 - c. If it is the last byte, then use the lower 4 bits to determine the sign. It must be one of 0xC, 0xD, or 0xF.
 - d. If not, multiply the lower digit by the multiplier and add the result to the total. Divide the multiplier by 10 for the next digit.

Working with a byte type in Java is problematic in this situation because it does not have an unsigned byte. For example, a byte containing 1001 0010 represents digits 9 and 2 in the packed decimal format. But in Java, it is interpreted as -110 because the most significant bit is used as a sign flag. Consequently, appropriate actions must be taken, especially examining the upper 4 bits of a byte. The following sample code retrieves the upper and lower digits as integer values:

```
int high = (byte & 0xFF) >> 4;  
int low  = byte & 15;
```

The first line retrieves a digit represented by the higher 4 bits of a byte. The AND (&) operation with 0xFF is required to mask out the sign bit of a byte first, then bits are shifted four position to the right to retrieve the corresponding digit. The second line retrieves the digit represented by the lower 4-bits of a byte. It simply performs the AND operation against 15 to mask out everything but the last 4 bits. For the complete implementation, refer to the source code of JRecord APIs.

3.3.2 Converting from Java integers to COBOL packed decimals

In this section, we describe an approach to convert Java integers to corresponding COBOL packed decimal formats. The following steps summarize the procedure required for the conversion process:

1. Allocate an array of bytes according to the size specification.
2. From the specified integer value, retrieve a list of digits and add 0s at the front if the specified value does not occupy the entire byte array. (For example, 23 becomes [0, 2, 3] for a 2-byte field. The last 4 bits are used for the sign).
3. Determine the sign according to the type (signed versus unsigned) and append to the list of digits. (For example, +23 becomes [0, 2, 3, C]).
4. For two digits at a time, pack the digits into a byte array.

The following Java sample code shows how to pack two digits into a byte:

```
byte = (byte) (top << 4);  
byte = (byte) (byte | bottom);
```

The variables *top* and *bottom* hold the corresponding digits to be packed into a byte (for example, 0 and 2 from [0, 2, 3, C]). The first step shifts everything by 4 bits and assigns it to a byte field. For example, if the top digit has a value of 5, the variable *byte* has a value of 0x50 as the result of the first step.

00000000 00000000 00000000 0000 <u>0101</u>	Digit 5 as a Java integer.
00000000 00000000 00000000 <u>0101</u> 0000	After shifting everything by 4 bits.
01010000	As a byte.

The second step performs the OR operation with the corresponding lower digit to pack two digits into a byte field. The following example shows what happens if digits 5 and 7 are packed into a byte:

00000000 00000000 00000000 <u>0101</u> 0000	Digit 5 from the previous step.
00000000 00000000 00000000 0000 <u>0111</u>	Digit 7 as a Java integer.
00000000 00000000 00000000 <u>0101</u> 0111	After the OR operation.
01010111	As a byte ² .

For the complete implementation of the conversion process, refer to the source code of JRecord APIs.

² Java bit-wise operators only work with integer numbers. Explicit castings are required for the conversion.

3.4 JRecord bean generator and supporting APIs

The read and write methods provided by the JRIO library return a record as an array of bytes. It is the application developer's responsibility to define the format of data within the record and perform appropriate conversion between the bytes and suitable Java types. Parsing of bytes into individual fields and conversion of bytes into Java types are an error-prone exercise and could easily lead to a problem in the application and performance degradation if poorly implemented. To address the issues and make the development effort somewhat easier, we provide Java APIs to perform the conversion between bytes and Java types, and an Eclipse plug-in that generates Java beans to represent records of an application in Java native types (we refer to the APIs and the plug-in as the *JRecord bean generator*). In this section, we provide an overview of the APIs and how to use the Eclipse plug-in. The utility APIs and the bean generator plug-in are intended to be a demonstration of techniques to perform conversions and a prototype of desirable development tools for developing Java applications for zSeries. Consequently, they are limited in their capabilities but can be extended further to accommodate other requirements.

3.4.1 JRecord utility APIs

The JRecordUtility class provides a set of utility methods that can be used to convert an array of bytes returned by the JRIO library read methods into various Java types, and convert them back into the corresponding byte array format. The following methods are provided:

intToCobolPacked	Converts a given Java integer value into COBOL packed decimal format.
cobolPackedToInt	Converts a given array of bytes holding a COBOL packed decimal format into Java integer value.
stringToBytes	Converts a given Java String into an array of bytes.
bytesToString	Converts a given array of bytes into a Java String format.

These methods can be used in any Java program to perform the conversion from the mainframe native types to Java types.

3.4.2 Generating a Java bean for a record

In addition to the utility methods described in the previous section, we provide an Eclipse plug-in that generates a Java bean class that represent fields within a record and provides parsing functionality to retrieve values for each field from an array of bytes returned by the JRIO library. Table 3-3 lists all the types supported by the plug-in.

Table 3-3 Supported types of JRecord

Type	Description
TEXT_FIELD	Text field. Converts the returned byte array into a string.
SIGNED_COBOL_PACKED_DECIMAL_FIELD	COBOL signed packed decimal. The number must be between 0 and Integer.MAX_VALUE.
UNSIGNED_COBOL_PACKED_DECIMAL_FIELD	COBOL unsigned packed decimal. The number must be between Integer.MIN_VALUE and Integer.MAX_VALUE.
BYTE_FIELD	Treats the field as unchanged bytes and simply copies byte-by-byte.

In this section, we describe how to use the plug-in by following a simple scenario. In this scenario, we have a VSAM data set containing customer records. We demonstrate how to read records from the VSAM data set using the JRIO library, and how to generate a Java bean class using the Eclipse plug-in to parse and manipulate the data.

Table 3-4 summarizes the fields within a customer record. It is a record consisting of a 10-byte key field (called *key*) and 27 bytes of data fields, which consist of first name (fName), last name (lName), and data of birth (year, month and day).

Table 3-4 Description of a customer record

Field name	Field type	Field size	Field description
key	byte	10	Key field for this record
fName	text	10	First name of a customer
lName	text	10	Last name of a customer
year	unsigned COBOL packed decimal	3	Data of birth of a customer (year)
month	unsigned COBOL packed decimal	2	Date of birth of a customer (month)
day	unsigned COBOL packed decimal	2	Date of birth of a customer (day)

Perform the following steps to generate a Java bean for this record:

1. Invoke the JRecord bean generation wizard from the context menu available in the Navigator view by selecting **JRecord Bean Generator** → **Bean Generation** (Figure 3-2).

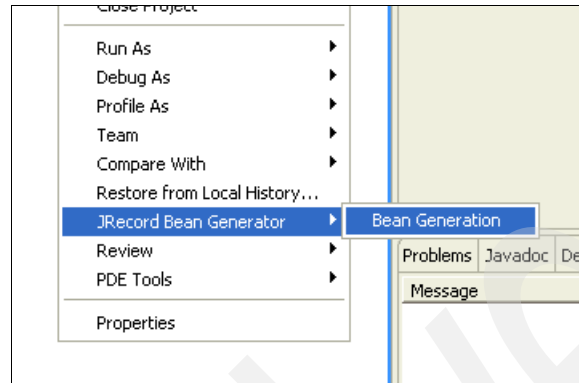


Figure 3-2 JRecord Java bean generator menu

When the Bean Generation menu item is selected, it invokes a wizard that guides you through the process of generating a Java bean class for the record.

2. Figure 3-3 on page 53 shows the first page of the wizard. Specify the name of the Java bean class on this page. The destination where the Java bean class is generated is also specified on this page. It can be generated to either the selected project or in a preferred directory. Generation to the selected project is the default selection. To generate to a different directory, click the **Select Directory** option and choose the desired directory. Once all information is entered, click **Next**.

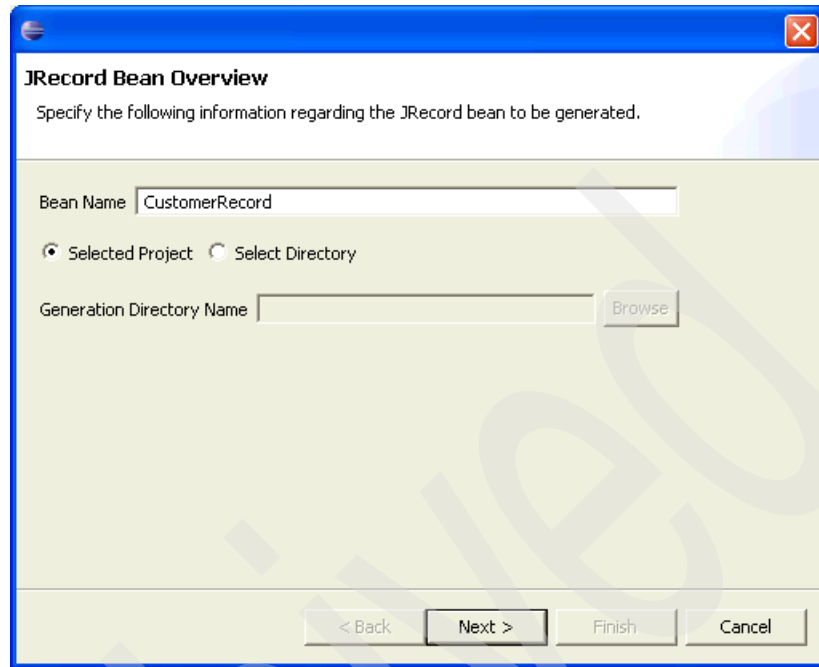


Figure 3-3 First page of Java bean generation wizard

3. Information about fields within a record is specified on the next page. Figure 3-4 shows the page, which lists all the fields in a record. For each field, the name, type, and size must be specified. You can add, modify, and delete fields here; click the button to perform the desired action. You also can change the order of the fields by highlighting a field and clicking the up and down arrows.

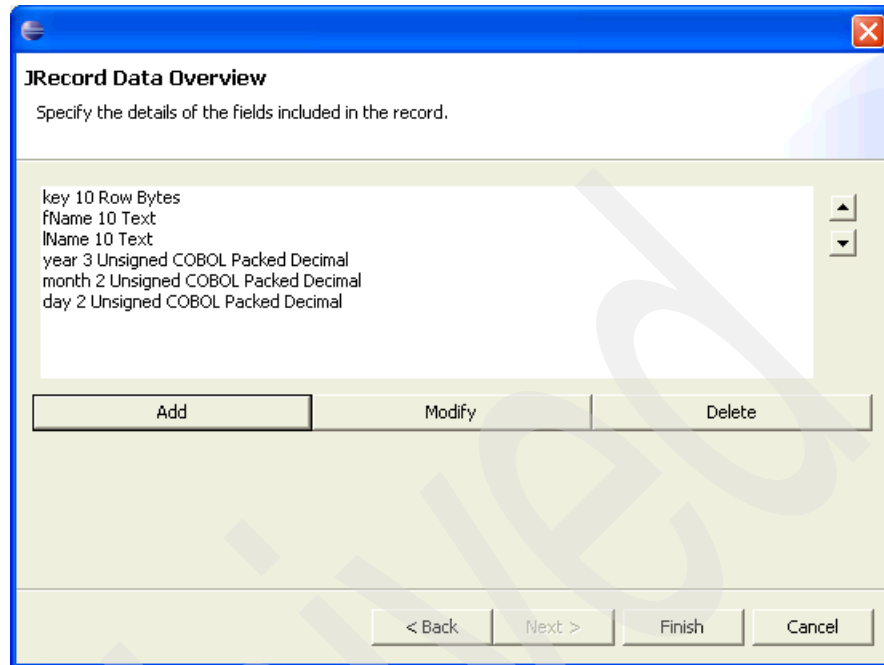


Figure 3-4 JRecord Java bean generation field details page

4. To add a new field, click **Add**. This returns a pop-up dialog box (Figure 3-5). Specify the details about the field in this box and click **OK**.

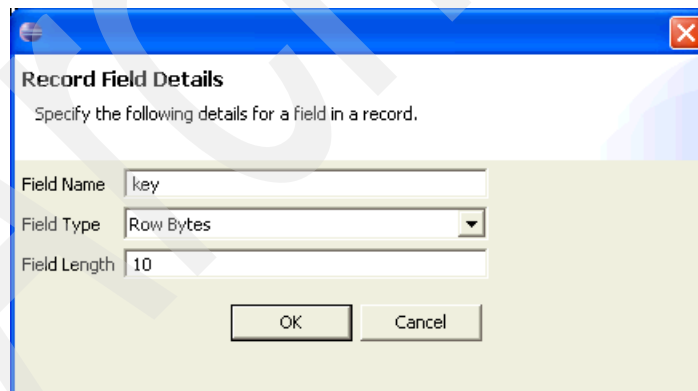


Figure 3-5 Popup dialog to specify the details of a field

5. Once all required information about a record is specified in the wizard, click **Finish** to generate a Java bean class to the specified location. The generated Java bean class should compile without any problem. To compile the source,

the JAR file of the JRecord bean generator library must be included in the classpath.

Example 3-5 shows the generated Java bean class called CustomerRecord.

Example 3-5 CustomerRecord Java bean class generated by JRecord Bean Generator

```
// Import required.
import com.ibm.itso.data.*;
import com.ibm.itso.data.exceptions.*;
import java.io.UnsupportedEncodingException;
// End of import statement.

**
 * This class is generated by JRecordBeanGenerator.
 */
public class CustomerRecord {
    // Beginning of member definitions.
    private byte[] key = new byte[10]; // Row byte field.
    private String fName; // Text field.
    private String lName; // Text field.
    private int year; // COBOL unsigned packed.
    private int month; // COBOL unsigned packed.
    private int day; // COBOL unsigned packed.
    // End of member definitions.

    // Beginning of getter/setter methods.

    // Row byte field getter method for key.
    public byte[] getkey() {
        return key;
    }

    // Row byte field setter method for key.
    public void setkey(byte[] b) {
        this.key = b;
    }

    // Text field getter method for fName.
    public String getfName() {
        return fName;
    }

    // Text field setter method for fName.
    public void setfName(String s) {
        this.fName = s;
    }
}
```

```

    }

    // Text field getter method for lName.
    public String getlName() {
        return lName;
    }

    // Text field setter method for lName.
    public void setlName(String s) {
        this.lName = s;
    }

    // COBOL unsigned packed decimal getter method for year.
    public int getyear() {
        return year;
    }

    // COBOL unsigned packed decimal setter method for year.
    public void setyear(int i) {
        this.year = i;
    }

    // COBOL unsigned packed decimal getter method for month.
    public int getmonth() {
        return month;
    }

    // COBOL unsigned packed decimal setter method for month.
    public void setmonth(int i) {
        this.month = i;
    }

    // COBOL unsigned packed decimal getter method for day.
    public int getday() {
        return day;
    }

    // COBOL unsigned packed decimal setter method for day.
    public void setday(int i) {
        this.day = i;
    }

    // End of getter/setter methods.

    // Method to convert bytes to data.

```

```

    public void bytesToData(byte[] data) throws
    UnsupportedEncodingException {
        // Beginning of data field retrieval.
        // Retrieve row bytes field key.
        System.arraycopy(data, 0, key, 0, 10);

        // Retrieve text field fName.
        fName = JRecordUtility.bytesToString(data, 10, 20, null);

        // Retrieve text field lName.
        lName = JRecordUtility.bytesToString(data, 20, 30, null);

        // Retrieve COBOL unsigned packed decimal field year.
        year = JRecordUtility.cobolPackedToInt(data, 30, 33);

        // Retrieve COBOL unsigned packed decimal field month.
        month = JRecordUtility.cobolPackedToInt(data, 33, 35);

        // Retrieve COBOL unsigned packed decimal field day.
        day = JRecordUtility.cobolPackedToInt(data, 35, 37);

        // End of data field retrieval
    }

    // Method to convert data into bytes.
    public byte[] dataToBytes()
        throws InvalidRecordException, UnsupportedEncodingException {
        byte[] tmp;
        byte[] data = new byte[37];

        // Set the row bytes field key.
        System.arraycopy(key, 0, data, 0, 10);

        // Set the text field fName.
        tmp = JRecordUtility.stringToBytes(fName, 10, null);
        System.arraycopy(tmp, 0, data, 10, tmp.length);

        // Set the text field lName.
        tmp = JRecordUtility.stringToBytes(lName, 10, null);
        System.arraycopy(tmp, 0, data, 20, tmp.length);

        // Set the unsigned COBOL packed decimal field year.
        tmp = JRecordUtility.intToCobolPacked(year, false, 3);
        System.arraycopy(tmp, 0, data, 30, tmp.length);
    }

```

```

// Set the unsigned COBOL packed decimal field month.
tmp = JRecordUtility.intToCobolPacked(month, false, 2);
System.arraycopy(tmp, 0, data, 33, tmp.length);

// Set the unsigned COBOL packed decimal field day.
tmp = JRecordUtility.intToCobolPacked(day, false, 2);
System.arraycopy(tmp, 0, data, 35, tmp.length);

return data;
}

// End of class definition generated by JRecordBeanGenerator.

```

The generated Java bean class has a set of getter and setter methods for each field defined in a record. These methods can be used to manipulate the data stored in a record using the Java types. In addition to getter and setter methods, the generated bean class has **bytesToData(...)** and **dataToBytes(...)** methods. The first method parses the data stored as an array of bytes which is returned by the JRIO library into the specified fields of a record. It should be used as follows:

```

CustomerRecord rec = new CustomerRecord();
rec.bytesToData(buffer);

```

The argument to **bytesToData(...)** method should contain the bytes returned by the JRIO library for a record. Once the data is parsed using the method, each field can be manipulated using getter and setter methods.

The second method, **dataToBytes(...)**, converts the data stored in a Java bean into an array of bytes which can be passed to the JRIO library. As a part of the method call, it converts Java types into appropriate byte format. The data to an array of bytes should be converted as follows:

```

byte[] data = dataToBytes();

```

The byte array returned from the method is ready to be used with the APIs provided by the JRIO library.

Example 3-6 shows a sample Java program that uses the Java bean class generated by the JRecord bean generator. It reads a record sequentially from a VSAM data set, creates an instance of CustomerRecord class, parses the record using the class, and displays the information.

Example 3-6 Java program with a Java bean generated by JRecord bean generator

```

import java.io.IOException;
import com.ibm.recordio.*;
public class VSAMExampleWithJRecord {

```

```

public static void main(String args[]) {

    try {
        String vFile = "//USER1.VSAM.FILE";
        // Must specify the record size of VSAM data explicitly if not 80.
        IRecordFile rf = RecordFile.getInstanceOf(vFile, 38,
            IConstants.JRIO_FIXED_MODE);
        IKeyedAccessRecordFile karf = new KeyedAccessRecordFile(rf,
            IConstants.JRIO_READ_WRITE_MODE);

        // Get a reference to the VSAM key file.
        IRecordFile keyFile = karf.getPrimaryIndex();

        byte[] buffer = new byte[38];
        while (karf.read(keyFile, buffer) != -1) {
            // Create Customer Record bean to manipulate the data in a
record.
            CustomerRecord rec = new CustomerRecord();

            // Parse the returned bytes.
rec.bytesToData(buffer);

            // Print the data.
            System.out.println("Family name = " + rec.getlName());
            System.out.println("First name = " + rec.getfName());
            System.out.println(rec.getyear() + "/" + rec.getmonth()
                + "/" + rec.getday());

            // Update the data.
rec.setyear(2005);
rec.setmonth(10);
rec.setday(7);
karf.update(keyFile, rec.dataToBytes());
        }

        // Close the output stream for VSAM file.
        karf.close();

    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

The generated Java bean classes rely on the functionality provided by JRecordUtility class, which is included in the JRecord bean generator Java archive file (JRecord.jar). To compile and run the program successfully, the jar file must be included in the CLASSPATH variable.

3.4.3 JRecord runtime library and plug-in installation

The JRecord runtime library is packed in a Java archive file (JAR) called JRecordRuntime.jar. To use the utility methods of the JRecordUtility class, the JAR file must be included in the CLASSPATH environmental variable. The JAR file should be included in the CLASSPATH variable as follows:

On Windows® platforms:

```
set CLASSPATH=%CLASSPATH%;JRecord.jar
```

On UNIX-like platforms:

```
CLASSPATH=${CLASSPATH}:JRecord.jar
export CLASSPATH
```

The JRecord bean generator plug-in is developed for the Eclipse 3.1 platform. It is distributed in a compressed file called JRecord Plugin.zip. It should be unzipped into the plugins directory in the Eclipse installation directory. It can be unzipped and installed as follows:

```
cd C:\eclipse
jar -xvf JRecordPlugin.zip
```

C:\eclipse should have a directory called plugins where the JRecord bean generator plug-in will be installed. When installing the plug-in, the Eclipse platform should be closed. Once the file is unzipped, the plug-in should be ready for use.

Tools and techniques

After a Java application has been developed, only a part of the work is done. The application must still be deployed, and eventually debugged, on the target platform, which in this discussion is z/OS. Debugging could be necessary even when extensive unit testing has been done in the IDE on the workstation. After all, some problems are caused by z/OS-specific settings or other environmental factors.

Fortunately, a lot of automation can be applied to the post-development activities like deploying the application on z/OS and performing subsequent (remote) debugging. We show in this chapter how Eclipse can be used to build and deploy stand-alone Java applications and then debug the applications while they are running on z/OS. The solution shown in this chapter is based on Eclipse SDK 3.1.

Although this chapter deals primarily with Eclipse, there are other tools available for debugging, such as the SDK built-in jdb tool, that can be run from the command line on z/OS.

There is a good amount of information available for J2EE application server (such as WebSphere) applications, but not much for the stand-alone type of applications that are the subject of this book.

Currently there is no direct IDE support for the build process for stand-alone Java applications for z/OS. Also, there is little information available on how to debug stand-alone Java applications running on z/OS from a Java IDE.

4.1 Introduction

Good tools can accelerate the process of developing stand-alone Java applications for z/OS. The Eclipse SDK (including various plug-ins) is one tool that provides strong support for such development activities, including the following:

- ▶ The Eclipse Ant support can be used to automate application packaging and deployment.
- ▶ The Eclipse Java debugger can be used to debug Java programs running on z/OS.

Eclipse is an open source project. It can be accessed at:

<http://www.eclipse.org>

4.2 Application build and deployment

Most of the stand-alone Java applications on z/OS are developed in a Java IDE. Normally, the compiled byte code classes are packaged into jar files and deployed to UNIX System Services (USS) on z/OS using ftp tools.

Unfortunately, jar files are not the only artifacts required. Besides the Java classes, there may be many other files needed for an application. These files include JCL, USS Shell script, properties file, configuration files, XML files, and others.

Moving these files to USS and managing the necessary code page conversion during ftp is a tedious and error-prone process. If the program can only be debugged and tested on z/OS (which is likely if the application is using some z/OS-specific resources), the application developer may have to ftp the revised version of the application to USS again and again. Automating the process can significantly ease application development.

The following sections discuss using Ant support in the Eclipse workbench to automate the packaging and deployment process of a stand-alone Java application running on z/OS.

4.2.1 Ant support in Eclipse

Apache Ant is an open source, Java-based build tool. For details about this tool, see:

<http://ant.apache.org>

The Eclipse Ant support allows you to create and run *Ant buildfiles* 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.

You can add classes to the Ant classpath, and add Ant tasks and types from the Ant Runtime preference page, by selecting **Window** → **Preferences** → **Ant** → **Runtime**.

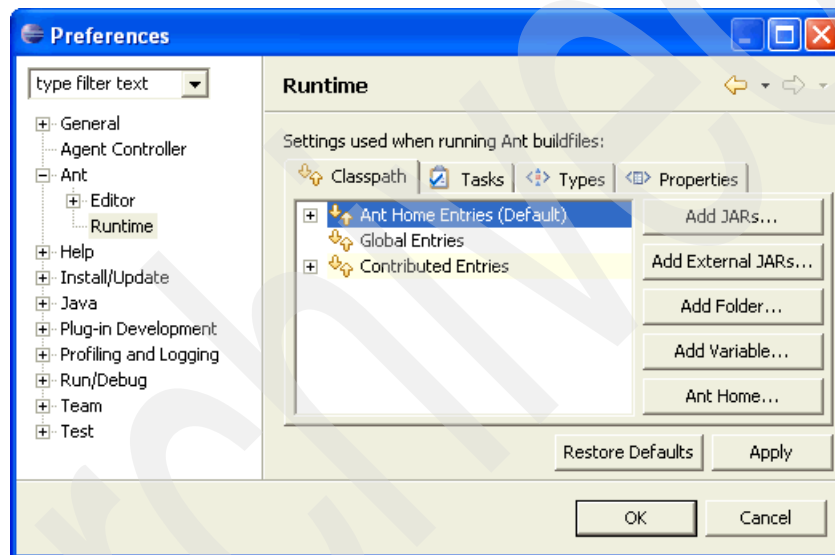


Figure 4-1 Ant Runtime setup

The Ant editor provides specialized features for editing Ant buildfiles. Associated with the editor is an Ant buildfile-specific outline view, which shows the structure of the Ant buildfile. It is updated as the user edits the buildfile.

To run an Ant buildfile in the workbench, select the buildfile in one of the navigation views.

Right-click the buildfile to bring up the context menu. Select **Run Ant**. The launch configuration dialog opens (Figure 4-2 on page 64).

Select one or more targets from the Targets tab. The order in which you select the items is the order in which they will run. You can find the order displayed in the Target execution order field at the bottom of the tab. You can change the

order of the targets by clicking the **Order** button. Ant is flexible, which lets you to run any targets in the Buildfile in virtually any reasonable sequence.

Optionally, configure options on the other tabs. For example, on the Main tab, type any required arguments in the Arguments field.

Click **Run**.

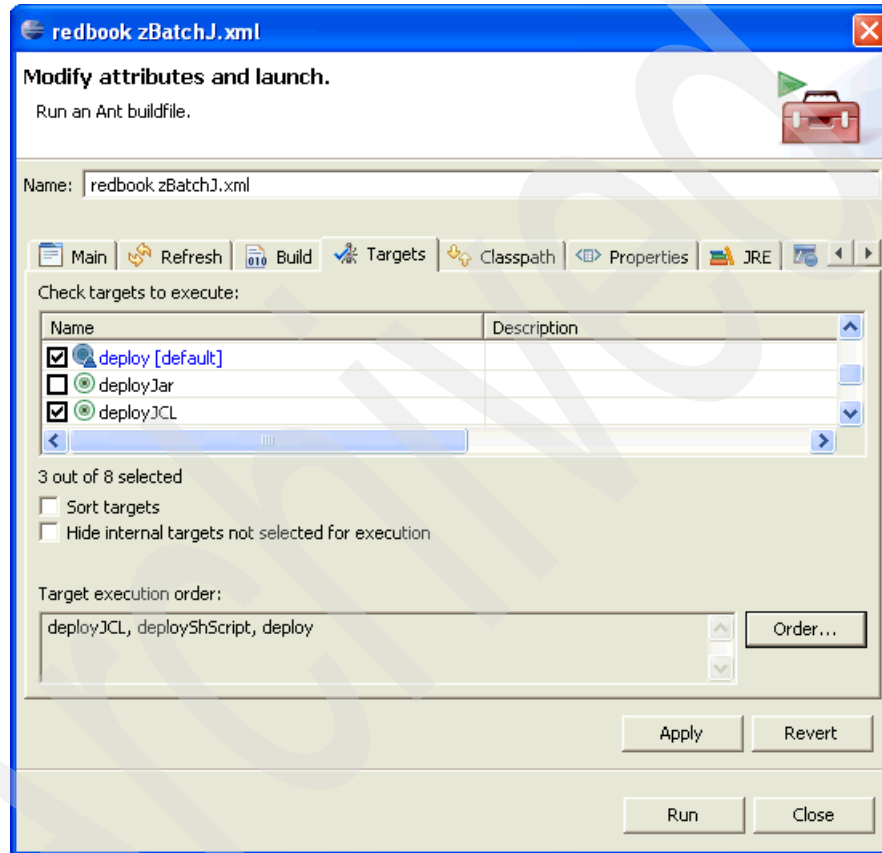


Figure 4-2 Ant Launch configuration dialog

The Ant buildfile will run on the selected targets. The console displays any applicable execution results as the buildfile runs unless the Capture output option on the Main tab is turned off.

These steps create a persisted launch configuration that can be directly invoked the next time. The configuration appears in the launch history under **Run** → **External Tools**, and is available in the launch configuration dialog that is opened by clicking **Run** → **External Tools** → **External Tools**.

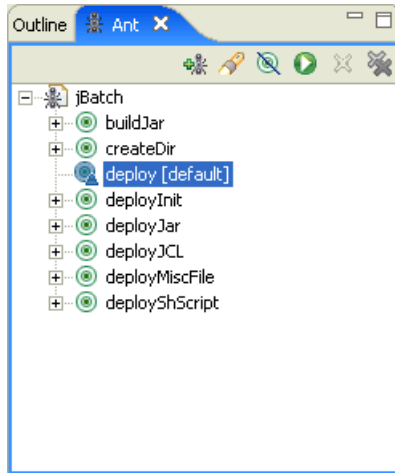



Figure 4-3 Ant view

You can also run an Ant buildfile from the Ant view shown in Figure 4-3. Highlight the target you want to run to select it. Click the  button. The Ant buildfile runs on the selected target.

4.2.2 Set up Ant ftp support in Eclipse

To enable application deployment to z/OS from the Eclipse workbench running on your workstation, the Ant ftp support in Eclipse must be set up.

Download commons-net-1.4.0.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-1.4.0.jar and jakarta-oro-2.0.8.jar from the zip files and put them into a directory called c:\ant_ftp_jars.

Select **Window** → **Preferences** → **Ant** → **Runtime**. Under the classpath tab, select **Ant Home Entries** and click the **Add External JARs** button. Specify the two jar files in the directory c:\ant_ftp_jars from the file selection dialog, and 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 Build and deployment Ant script

Ant can automate a build process by taking a specification file and performing a sequence of operations based on the content of that file and the state of the file system.

As discussed before, packaging and deploying a Java application to z/OS consists of a sequence of operations such as compiling, creating a jar file, and ftping files. This build process can be automated using an Ant buildfile like the one in Example 4-1, called zBatchJ.xml.

Example 4-1 Build and Deployment Ant Script zBatchJ.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<project name="jBatch" default="deploy">
  <!-- This ant script deploys a batch Java applications to z/OS
        using ftp.
  -->

  <property file="./zBatchJ.properties" />

  <target name="deployInit">
    <mkdir dir="${project.jar.dir}" />
  </target>

  <target name="createDir" >
    <mkdir dir="${project.jcl.dir}" />
    <mkdir dir="${project.shscript.dir}" />
    <mkdir dir="${project.misc.dir}" />
  </target>

  <target name="buildJar" depends="deployInit" >
    <jar destfile="${project.jar.dir}/${project.app.jar}" >
      <fileset dir=".">
        <exclude name="**/*.java"/>
        <exclude name="**/*.jar" />
      </fileset>
    </jar>
  </target>

  <target name="deploy" depends="buildJar, deployJCL, deployShScript,
    deployMiscFile" />

  <target name="deployJar" depends="buildJar" >
    <echo message="Sending jar files to ${zOS.hostname}: ${zOS.jar.dir}
    ..."/>
    <ftp server="${zOS.hostname}"
      userid="${zOS.ftp.userid}"
      password="${zOS.ftp.password}"
```

```

        remotedir="${zOS.jar.dir}"
        action="mkdir"
        verbose="yes" >
    </ftp>
    <ftp server="${zOS.hostname}"
        userid="${zOS.ftp.userid}"
        password="${zOS.ftp.password}"
        remotedir="${zOS.jar.dir}"
        depends="no"
        binary="yes"
        umask="002"
        verbose="yes" >
        <fileset dir="JARs" includes="" casesensitive="yes" />
    </ftp>
</target>

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

<target name="deployShScript" >
    <echo message="Sending Shell Scripts to ${zOS.hostname} :
${zOS.script.dir} ..."/>
    <ftp server="${zOS.hostname}"
        userid="${zOS.ftp.userid}"
        password="${zOS.ftp.password}"
        remotedir="${zOS.script.dir}"
        action="mkdir"
        verbose="yes" >
    </ftp>
    <ftp server="${zOS.hostname}"
        userid="${zOS.ftp.userid}"
        password="${zOS.ftp.password}"
        remotedir="${zOS.script.dir}"
        binary="no"
        verbose="yes"
        depends="no" >

```

```

        <fileset dir="${project.shscript.dir}" >
            <include name="*" />
        </fileset>
    </ftp>
</target>

<target name="deployMiscFile" >
    <echo message="Sending Misc Files to ${zOS.hostname} : ${zOS.misc.dir}
..."/>
    <ftp server="${zOS.hostname}"
        userid="${zOS.ftp.userid}"
        password="${zOS.ftp.password}"
        remotedir="${zOS.misc.dir}"
        action="mkdir"
        verbose="yes" >
    </ftp>
    <ftp server="${zOS.hostname}"
        userid="${zOS.ftp.userid}"
        password="${zOS.ftp.password}"
        remotedir="${zOS.misc.dir}"
        binary="no"
        verbose="yes"
        depends="no" >
        <fileset dir="${project.misc.dir}" >
            <include name="*" />
        </fileset>
    </ftp>
</target>

</project>

```

This Ant script is designed to be used within a Java project created in the Eclipse workbench.

The following eight targets are in this buildfile:

- ▶ **deployInit** defines the operation creating a folder \${project.Jar.dir} to hold the jar file of the application if the folder does not exist.
- ▶ **createDir** defines the operation creating a set of folders to hold the application's different types of non-java files. This target should be run during the application project creation in the Eclipse workspace instead of at deployment time. The target creates three folders:
 - \${project.JCL.dir} is created to hold JCL and Procs for launching the application on z/OS.
 - \${project.shscript.dir} is created to hold a USS Shell script for environment setup and Java program launching.

- `${project.misc.dir}` is created to hold other files for the application. The other files could be XML files, properties files, configuration files, and so forth. One folder may not be enough to hold these files. Furthermore, these files may need to be stored in a group hierarchy structure of directories. So, feel free to customize this part of the Ant script.
- ▶ **buildJar** defines the operation of packaging all the class files in a specific Java project into a Jar file in the `${project.jar.dir}` directory created by the `deployInit` target. The Jar file name is `${project.app.jar}`. This target depends on `deployInit`. You can customize this target to include or exclude files in the Jar file.
- ▶ **deploy** target is the default target of the Ant script. It depends on the `deployJar`, `deployJCL`, `deployShScript` and `deployMiscFile` targets. If this target is selected to run, Ant runs all dependent targets first. Since the `deploy` is the default target, Ant runs on it whenever there is no target specified to run.
- ▶ **deployJar** defines the operation of sending the Jar file built by `buildJar` to the USS directory `${zOS.jar.dir}` using Ant ftp. There are two ftp steps defined: the first step creates the directory in USS, the second step sends the Jar file in binary mode. This target depends on `buildJar`.
- ▶ **deployJCL** defines the operation of sending the JCL/Proc to a PDS on z/OS. The JCL/Proc becomes a PDS member. The ftp is in text mode.
- ▶ **deployShScript** defines the operation of sending the shell script created for the project to the USS directory `${zOS.script.dir}`. There are two steps defined in the operation as well. The first one is to create the directory in USS, the second one does the ftp work. The ftp should be done in text mode.

Note: The `deployShScript` target may need an extra step to change the shell script access mode in USS to make the shell script executable.

- ▶ **deployMiscFile** defines the operation of sending other non-Java files to the USS directory `${zOS.misc.dir}`. It also has two steps defined. One handles directory creation. The other one handles the file transfer. The file transfer mode needs to be considered based on the JVM file encoding. It must to be determined application by application.

Note: The `deployMiscFile` target may need an extra step to control the file access mode in USS through `chmod`.

The variables in the regular expressions of the Ant script are defined in the properties file `zBatchJ.properties` (Example 4-2 on page 70).

Example 4-2 Properties file zBatchJ.properties

```
zOS.hostname=9.12.4.126
zOS.jar.dir=/u/alex03/app/jar
zOS.script.dir=/u/alex03/app/script
zOS.misc.dir=/u/alex03/app/misc
zOS.JCL.dsn='ALEX03.JCL.LIB'
zOS.ftp.userid=user1
zOS.ftp.password=password
project.Jar.dir=JARs
project.JCL.dir=JCL
project.misc.dir=MiscFiles
project.app.jar=debugdemo.jar
```

The Ant buildfile is more generic and flexible when using a properties file. The properties file provides a centralized place to control the customization of the Ant script. In most cases you only need to adjust the properties file and use the Ant buildfile without any change.

4.2.4 Use the ANT buildfile

This section describes how to use the Ant buildfile in an application development project.

Create a new Java project called zBatchJava in the Eclipse workbench. Import zBatchJ.xml and zBatchJ.properties into the project. Adjust the properties values in the zBatchJ.properties file to fit your application's requirements.

Right-click zBatchJ.xml to bring up the context menu. Select **Run As** → **2 Ant Build**. The launch configuration dialog opens. Run target **createDir** from the launch configuration dialog. Figure 4-4 shows the result of running the **createDir** target.

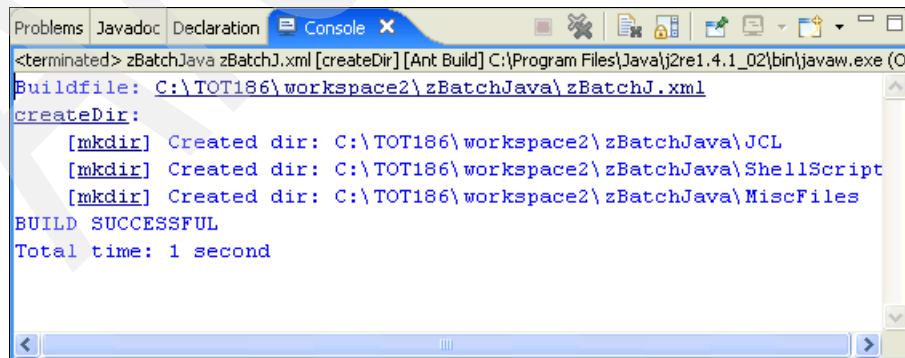


Figure 4-4 Running result of createDir target

Refresh the project in the Package Explorer view. The project zBatchJava has a structure like that shown in Figure 4-5.

A simple z/OS stand-alone Java application project structure is introduced here to support the deployment automation. Once the project structure is established, you can start Java program coding. All JCLs for the application should be created under the JCL directory. All shell scripts go under the ShellScript directory. Other files go to the MiscFiles directory.

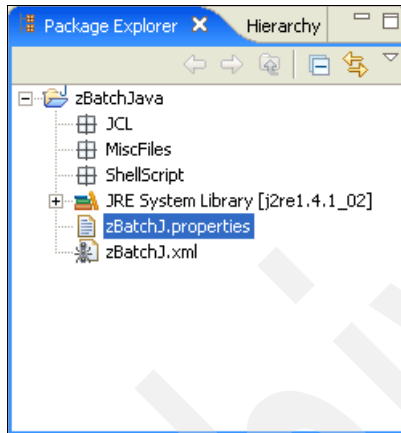


Figure 4-5 z/OS stand-alone Java application project

Once the application is ready for deployment, just run the deploy target either from the launch configuration dialog or from the Ant view in the Eclipse workbench. The deployment result can be found in the workbench console.

Whenever the application has any changes, you can run either the deploy target or just one of the deployment targets. There is no need to export the Java project into a Jar file, use a separate text editor to create Shell scripts, and manually manage and transfer files using an ftp tool anymore.

Therefore, building and deploying a stand-alone Java application for z/OS from your Eclipse workbench has become fairly easy.

The second part of this chapter uses the Ant buildfile described here to build and deploy a sample Java application to z/OS to demonstrate remote debugging.

4.2.5 Further discussion

It is possible to develop an Eclipse plug-in that introduces a new project type: *z/OS Stand-alone Java Application project*. A new project wizard can easily

populate the properties file, create a project structure, and automatically generate the Ant buildfile from a template.

The context menu of the project can contain a new menu item called *Deploy*. Whenever it is clicked, a dialog opens to configure how to run the deploy. Then the configuration can be kept in the deploy history list for direct invocation later.

If the plug-in provided a JCL and USS shell script generation wizard and allowed submitting and checking z/OS batch jobs from the Eclipse workbench, the solution would be even more elegant. As it currently exists, it should dramatically improve the productivity of developing stand-alone Java application running on z/OS.

4.3 Debugging

The following sections describe how to debug stand-alone Java programs developed to run on z/OS.

4.3.1 Remote debugging

There is no Java IDE running on z/OS. Therefore, remote debugging is particularly important when the Java program accesses resources that are only accessible on z/OS, such as sequential data sets, VSAM files, and so forth. It is extremely useful, as well, when you try to fix problems that occur only on z/OS even when the Java program does not access any z/OS specific resources.

The Eclipse Java debugger is designed as a client/server architecture. It allows you to launch your stand-alone Java application on z/OS and debug it from the Eclipse workbench running on your workstation.

To debug a Java application running on z/OS remotely from Eclipse, the program should be launched on z/OS in the debug mode so that the JVM is waiting for a connection from your debugger. The basic steps are as follows:

1. Make sure you are building your Java programs with the necessary debug information. It can be controlled in Eclipse by configuring preferences.
2. Deploy your Java application to the target z/OS run-time environment. This involves copying files such as your class or jar files to certain USS directories, setting up necessary environment variables such as classpath, libpath, and so forth, and preparing running shell scripts and JCLs.
3. Start your Java application on z/OS using the appropriate VM arguments to specify debug mode and the communication port for the debugger.

4. Start the debugger using a remote debugging configuration and specify the address and port of the target machine running z/OS.

The following sections walk you through these four steps using an example.

4.3.2 Preparing the Java program

ProcessTrans.java is a simple Java program that reads data from a z/OS data set using the IBM JRIO API. Since the program reads a z/OS data set, we use it as the sample to demonstrate remote debugging of a stand-alone Java application running on z/OS. The program's source code is included in the additional material for this book.

Create a new project called *redbook* in the Eclipse workbench and import the Ant script *zBatchJ.xml* and the properties file *zBatchJ.properties* that were created earlier in this chapter¹. Figure 4-6 depicts the project in Eclipse Package Explorer view.

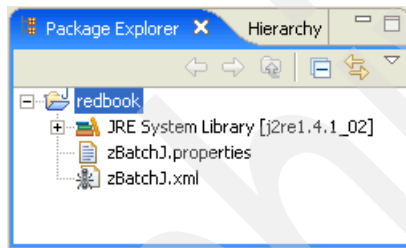


Figure 4-6 Redbook project (1)

Right-click *zBatchJ.xml*, and in the context menu select **Run As** → **2 Ant Build**. Run the target *createDir* from the launch configuration dialog. Several directories are added to the project.

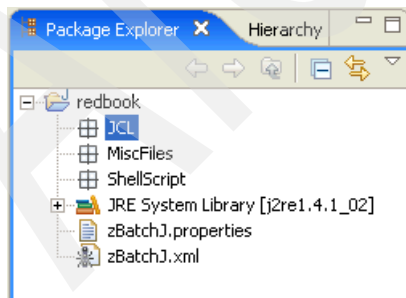


Figure 4-7 Redbook project (2)

¹ You can also import those files directly from the additional material for this book, in case you have not created them yourself.

Import the ProcessTrans.jar into the redbook project. To cleanly compile the program, add recjava.jar and recordio.jar to the project's build path. The jar files can be found in the IBM SDK for z/OS Java 2 Technology Edition, V1.4 and IBM SDK for z/OS Java 2 Technology Edition, V5.

Note: Be aware that recjava.jar and recordio.jar files are part of the IBM JRIO. They can only be used meaningfully on z/OS. We add these files to the project just for the sample Java program compilation.

Figure 4-8 shows the project now.

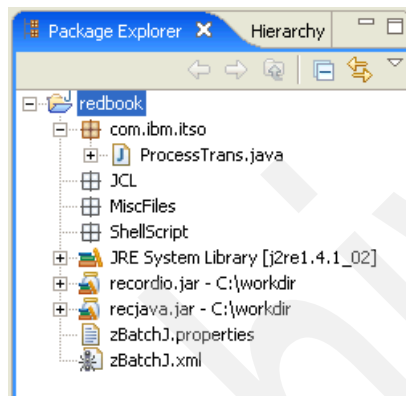


Figure 4-8 Redbook project (3)

To build the program for debugging, 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 is displayed (Figure 4-9).

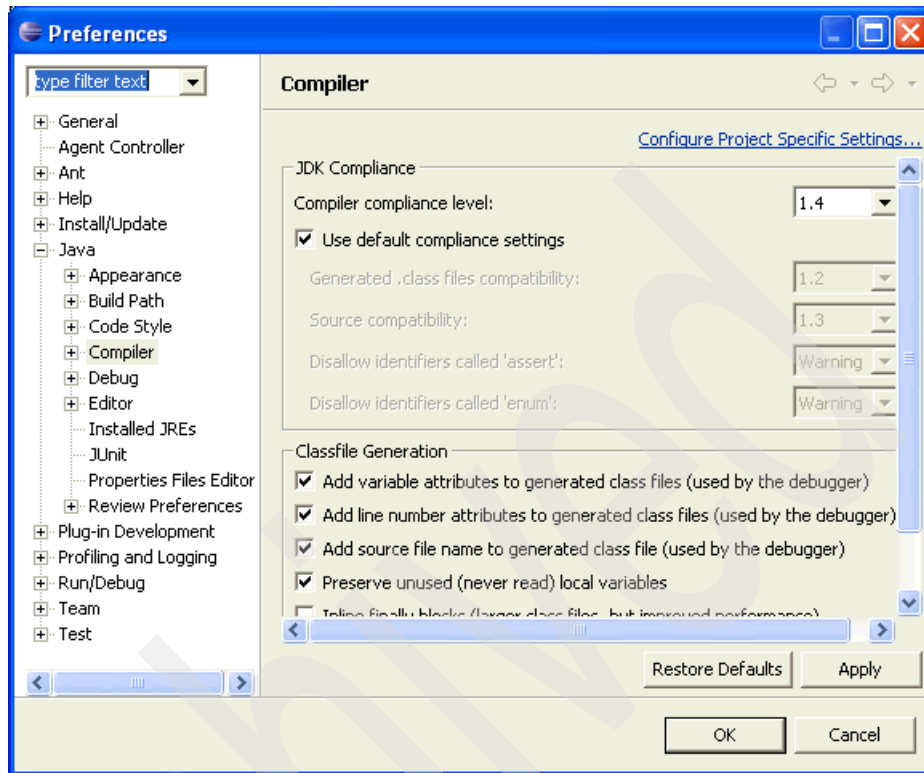


Figure 4-9 Preference dialog

Under **Java** → **Compiler**, the first three Classfile Generation options should be checked. Re-build the project. Now the program contains all necessary information for debugging.

4.3.3 Creating a shell script and JCL

On z/OS, a Java program can be launched either as a USS process using a shell script or as a batch job using a JCL/Proc. No matter how the Java program is launched, it can be debugged remotely.

The shell script used to launch a Java program in USS is the same as the shell script used on any other UNIX platforms. It sets the environment variables such as PATH, LIBPATH, CLASSPATH, and so forth, and uses the famous JVM launcher java to launch the Java program.

The sample program ProcessTrans.java runs from USS directory /u/alex03. All Java classes created in the project are packed into debugdemo.jar. Example 4-3

shows a shell script used to launch the sample program. Create the shell script debug.sh under the shell script folder in the redbook project.

Example 4-3 Shell Script debug.sh

```
#set up the PATH for JVM launcher
PATH=/bin:/usr/lpp/java/J1.4/bin:/u/alex03:$PATH
export PATH
#set up the CLASSPATH for debugging sample
CLASSPATH=/u/alex03/app/jar/debugdemo.jar:$CLASSPATH
export CLASSPATH
#launch the JVM for debugging
java -Xdebug -Xrunjdwp:transport=dt_socket,server=y,address=8000
com.ibm.itso.ProcessTrans 'ALEX02.DATA(MYDATA2)'
```

The last two lines in the example are actually one command line. -Xdebug and -Xrunjdwp 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 non-existent server because the default value of the server option is n. In the shell script in Example 4-3, the JVM starts as the debugging server that listens on port 8000.

Other options and explanations for -Xrunjdwp are in Table 4-1.

Table 4-1 -Xrunjdwp 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 n	Use C Runtime malloc/free?	n

A JCL/Proc is needed to launch a Java program as a batch job. More detailed information on launching Java programs using JCL/Proc is in Chapter 2, "Job

management” on page 11. Create the JCL as shown in Example 4-4 under the JCL folder in the redbook project with name *JavaRun*.

Example 4-4 JCL JavaRun

```
//ALEX03P JOB (APC),'ALEX03',REGION=0M,
//          CLASS=A,MSGCLASS=X,NOTIFY=&SYSUID
//* -----
//* Delete the output file in USS
//* -----
//DELETE   EXEC PGM=IEFBR14
//STDOUT   DD PATH='/u/alex03/java.stdout',
//          PATHOPTS=(OCREAT,OWRONLY),
//          PATHMODE=SIRWXU,
//          PATHDISP=(DELETE)
//STDERR   DD PATH='/u/alex03/java.stderr',
//          PATHOPTS=(OCREAT,OWRONLY),
//          PATHMODE=SIRWXU,
//          PATHDISP=(DELETE)
//* -----
//* Launch Java program
//* -----
//RUN      EXEC PGM=BPXBATCH,
//          PARM='sh /u/alex03/app/script/debug.sh'
//SYSPRINT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//STDOUT   DD PATH='/u/alex03/java.stdout',
//          PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//          PATHMODE=SIRWXU
//STDERR   DD PATH='/u/alex03/java.stderr',
//          PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//          PATHMODE=SIRWXU
//STDENV   DD DUMMY
//* -----
//* Copy the output of Java program
//* -----
// IF (RC EQ 0) THEN
//COMPOUT   EXEC PGM=IKJEFT01,DYNAMNBR=300,COND=EVEN
//SYSTSPRT DD SYSOUT=*
//HFSOUT   DD PATH='/u/alex03/java.stdout'
//HFSERR    DD PATH='/u/alex03/java.stderr'
//STDOUTL  DD SYSOUT=*,DCB=(RECFM=VB,LRECL=133,BLKSIZE=137)
//STDERRL  DD SYSOUT=*,DCB=(RECFM=VB,LRECL=133,BLKSIZE=137)
//SYSPRINT DD SYSOUT=*
//SYSTSIN  DD DATA,DLM='>'
//          ocopy indd(HFSOUT) outdd(STDOUTL)
//          ocopy indd(HFSERR) outdd(STDERRL)
//          />
// ENDIF
```

```

/* -----
/* Delete the output file in USS
/* -----
// IF (RC EQ 0) THEN
//DELETE2 EXEC PGM=IEFBR14
//STDOUT DD PATH='/u/alex03/java.stdout',
//        PATHOPTS=(OCREAT,OWRONLY),
//        PATHMODE=SIRWXU,
//        PATHDISP=(DELETE)
//STDERR DD PATH='/u/alex03/java.stderr',
//        PATHOPTS=(OCREAT,OWRONLY),
//        PATHMODE=SIRWXU,
//        PATHDISP=(DELETE)
// ENDIF

```

Essentially, the JCL runs the shell script debug.sh using the utility BPXBATCH.

The next section describes how to deploy all the artifacts created to z/OS and how to launch the sample program on z/OS.

4.3.4 Deploying and starting a Java application

The redbook project should appear in the Eclipse Package Explorer view as shown in Figure 4-10.

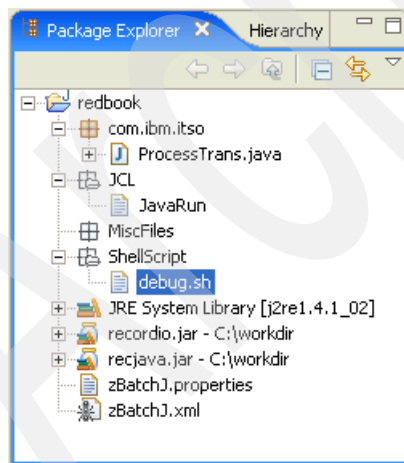


Figure 4-10 Redbook project ready for deployment

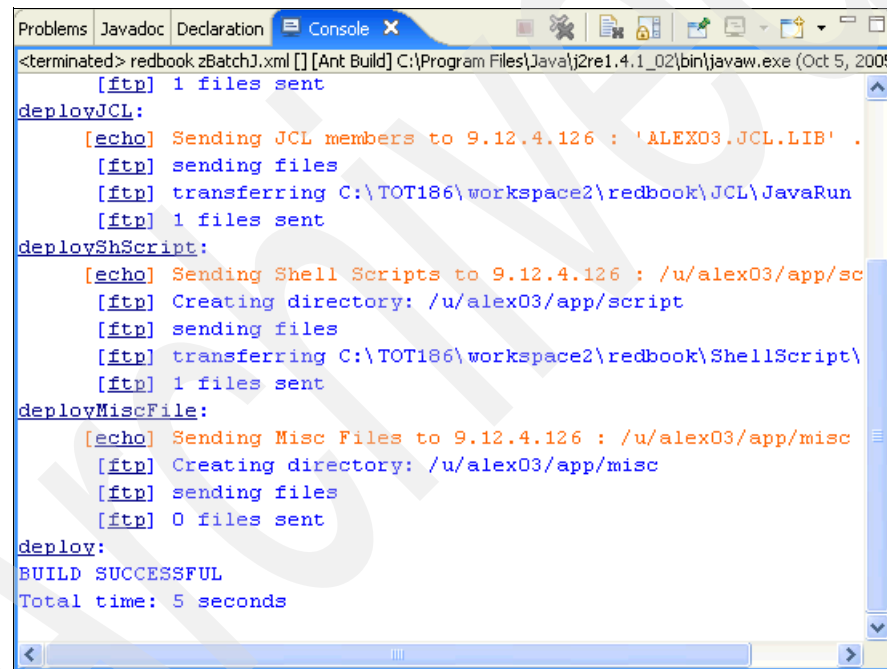
Right-click `zBatchJ.xml` in the `redbook` project to bring up the context menu in the Package Explorer view. In the context menu, select **Run As** → **2 Ant Build**, then run the target `deploy` from the launch configuration dialog.

As shown in Figure 4-11, the Ant Script sends the artifacts of the redbook project to the destinations on z/OS that are listed in Table 4-2.

Table 4-2 Deployment destinations

Artifact name	Destination
debugdemo.jar	/u/alex03/app/jar
debug.sh	/u/alex03/app/script
JavaRun	ALEX03.JCL.LIB(JAVARUN)

Once the application deployment is done, the next step is to launch the Java program on z/OS.



```
<terminated> redbook zBatchJ.xml [] [Ant Build] C:\Program Files\Java\j2re1.4.1_02\bin\javaw.exe (Oct 5, 2009)
[ftp] 1 files sent
deployJCL:
[echo] Sending JCL members to 9.12.4.126 : 'ALEX03.JCL.LIB' .
[ftp] sending files
[ftp] transferring C:\TOT186\workspace2\redbook\JCL\JavaRun
[ftp] 1 files sent
deployShScript:
[echo] Sending Shell Scripts to 9.12.4.126 : /u/alex03/app/script
[ftp] Creating directory: /u/alex03/app/script
[ftp] sending files
[ftp] transferring C:\TOT186\workspace2\redbook\ShellScript\
[ftp] 1 files sent
deployMiscFile:
[echo] Sending Misc Files to 9.12.4.126 : /u/alex03/app/misc
[ftp] Creating directory: /u/alex03/app/misc
[ftp] sending files
[ftp] 0 files sent
deploy:
BUILD SUCCESSFUL
Total time: 5 seconds
```

Figure 4-11 Deployment Result

Use the TSO command **OMVS** to establish a USS session. Change the current directory to /u/alex03/app/script, and then run debug.sh. The JVM is listening for the debugger now.

As shown in Figure 4-12 on page 80, process with PID 16777420 that is launched for remote debugging.

Note: When the shell script runs in foreground, it blocks the current USS session. If you don't want the USS session to be blocked, the shell script should be run in background.

```

Session A - [24 x 80]
File Edit View Communication Actions Window Help

ALEX03:/u/alex03/app/script: >. debug.sh&
Y1" 33554630
ALEX03:/u/alex03/app/script: >ps -Af
      UID      PID      PPID  C   STIME TTY      TIME CMD
ALEX03 33554540 33554548 - 10:27:43 tttyp0001 0:00 sh -L
ALEX03 33554548 33555151 - 10:27:37 ?      0:00 rlogind2 -a -m xterm/3
8400 alex03 9.12.12.81
ALEX03 67109056 1 - 16:14:31 ?      0:00 OMVS
ALEX03 33554630 199 - 16:17:11 tttyp0005 0:00 sh -L
ALEX03 199 67109056 - 16:14:31 tttyp0005 0:00 sh -L
ALEX03 16777420 33554630 - 16:17:11 tttyp0005 0:00 java -Xdebug -Xrunjdw
:transport=dt_socket,server=y,address=8000 com.ibm.itso.
ALEX03 50331857 199 - 16:17:20 tttyp0005 0:00 ps -Af
ALEX03:/u/alex03/app/script: >

===>
                                INPUT
ESC=⌘ 1=Help 2=SubCmd 3=HlpRetrn 4=Top 5=Bottom 6=TSO
7=BackScr 8=Scroll 9=NextSess 10=Refresh 11=FwdRetr 12=Retrieve
MA a 21/007
Connected to remote server/host 9.12.4.126 using lu/p

```

Figure 4-12 Launch JVM for remote debugging in USS

The Java program can be started as a batch job as well. The JCL JAVARUN is submitted to the Job Entry System(JES) from ISPF (Figure 4-13 on page 81).

Note: To run a shell script using BPXBATCH, the shell script has to be executable. Otherwise, the submitted job fails with return code 0126. Check the shell script, and make changes using the **chmod** command if necessary.

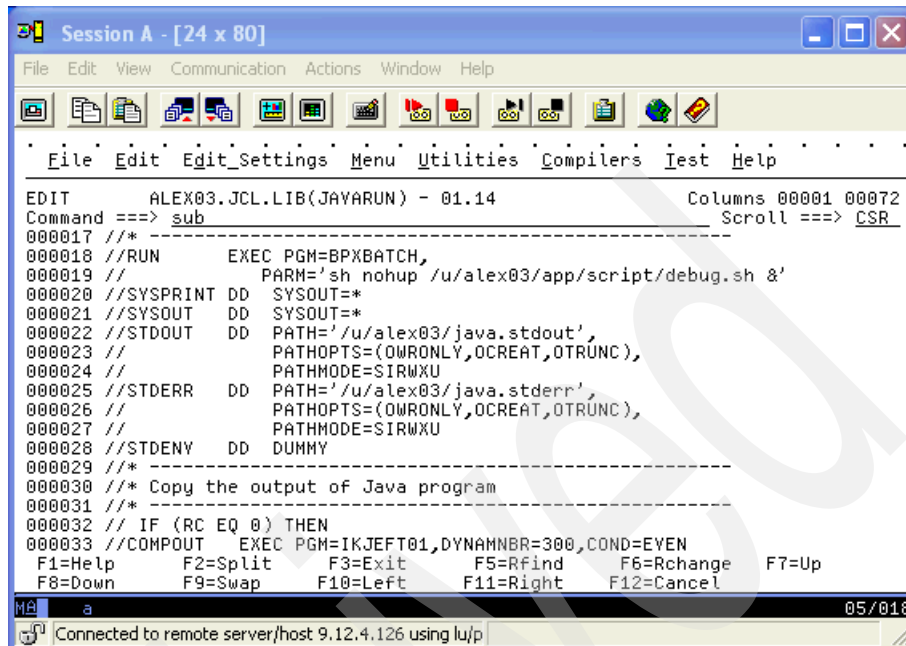


Figure 4-13 Submitting a Java batch job from ISPF

Figure 4-14 on page 82 is the SDSF process panel. You can check the job just submitted in this panel. The processes started in USS can be monitored from this panel as well.

There are three jobs involved to launch the JVM and the sample program: ALEX03P, ALEX03P5 and ALEX03P6. ALEX03P is the job submitted using the JCL; the other two are automatically created by BPXBATCH. As a child of ALEX03P, ALEX03P5 opens a USS session and forks a child process ALEX03P6. The ALEX03P6 launches the JVM and Java classes. More information about BPXBATCH can be found in Chapter 2, “Job management” on page 11.

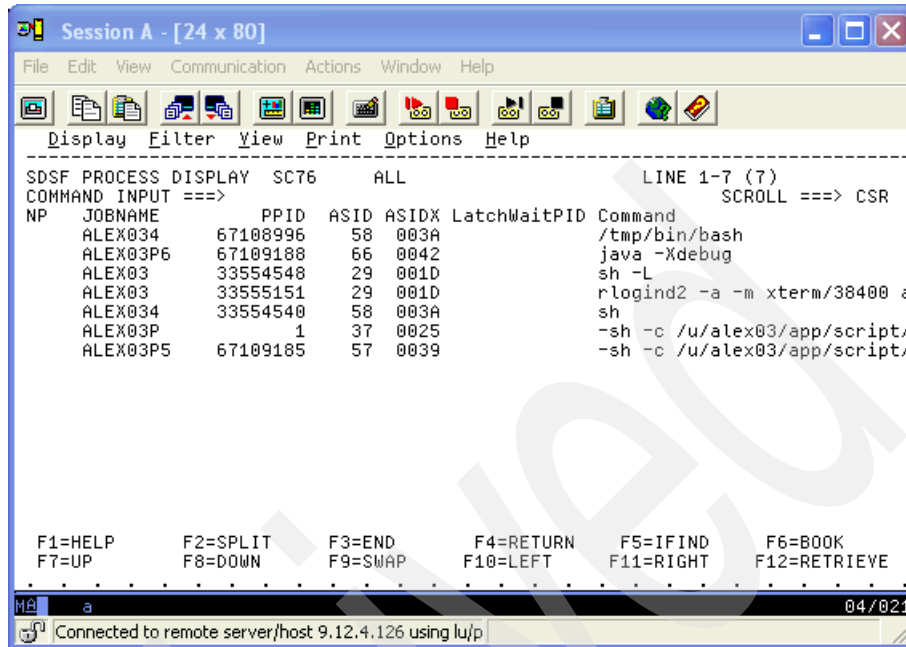


Figure 4-14 SDSF process panel

4.3.5 Work with the debugger in the Eclipse workbench

Eclipse has a build-in debugger that is designed to debug different types of Java applications such as Eclipse applications, Java Applets, Java applications, JUnits, JUnit Plug-in Tests, SWT applications, and remote Java applications. This section discusses how to use the Eclipse debugger to debug the sample program running on z/OS.

Open the Eclipse workbench. From the menu bar select **Window** → **Open Perspective** → **Debug** to switch to the Debug perspective (Figure 4-15 on page 83).

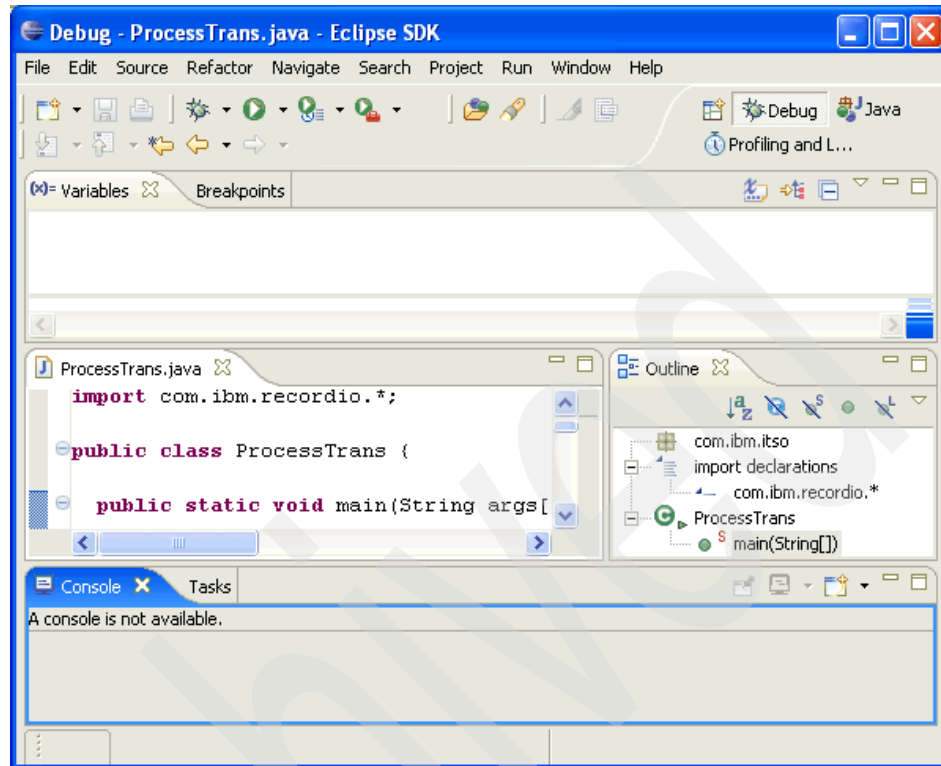



Figure 4-15 Debug perspective

Click the  icon and select **Debug**. The Debug wizard opens (Figure 4-16 on page 84).

The Remote Java Application launch configuration is used when debugging an application that is running on a remote JVM. Since the application is started on the remote system, the launch configuration does not specify the usual information about the JRE, program arguments, or JVM arguments. Instead, information about connecting to the application is supplied.

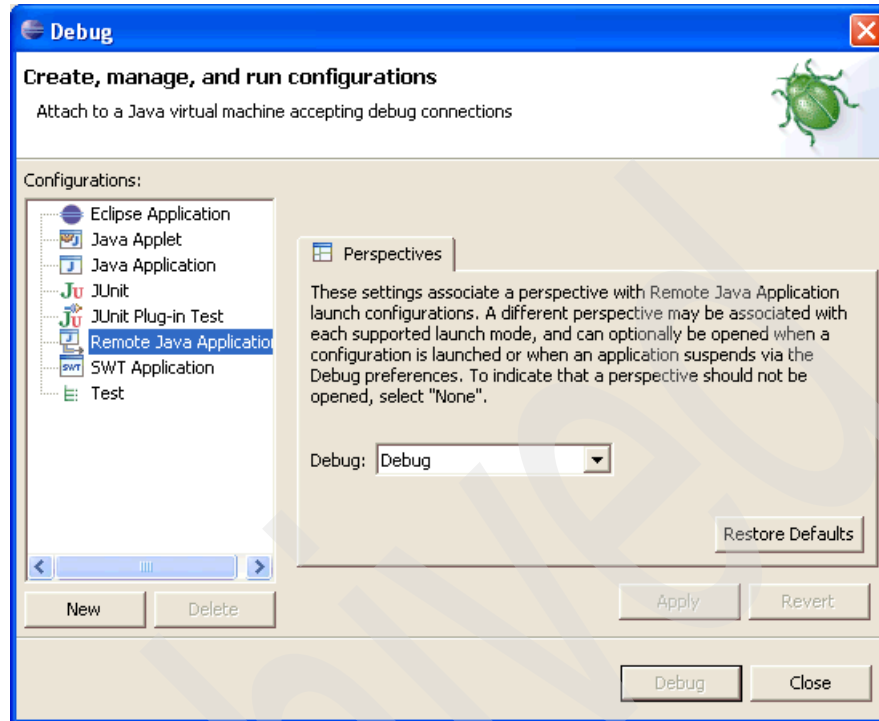


Figure 4-16 Debug wizard

Double-click **Remote Java Application** or click the **New** button. A new debugger configuration is created; details are shown in the right pane of the debug wizard (Figure 4-17 on page 85). You can change the name in the Name input text box to anything you want. Debug_ZOS is used for the demonstration.

Under the Connect tab, connection-related information for the debug configuration is displayed.

The Project input text box specifies the project of the configuration. To view a list of available projects, click the **Browse** button. The Project selection dialog box opens. Select the redbook project and click **OK**.

By default the connection type is Standard(Socket Attach). The JVM is started on z/OS using option -Xrunjdpw:transport=dt_socket,server=y,address=8000; therefore, the debugger must connect to the JVM by Socket Attach. There is no need to change the connection type.

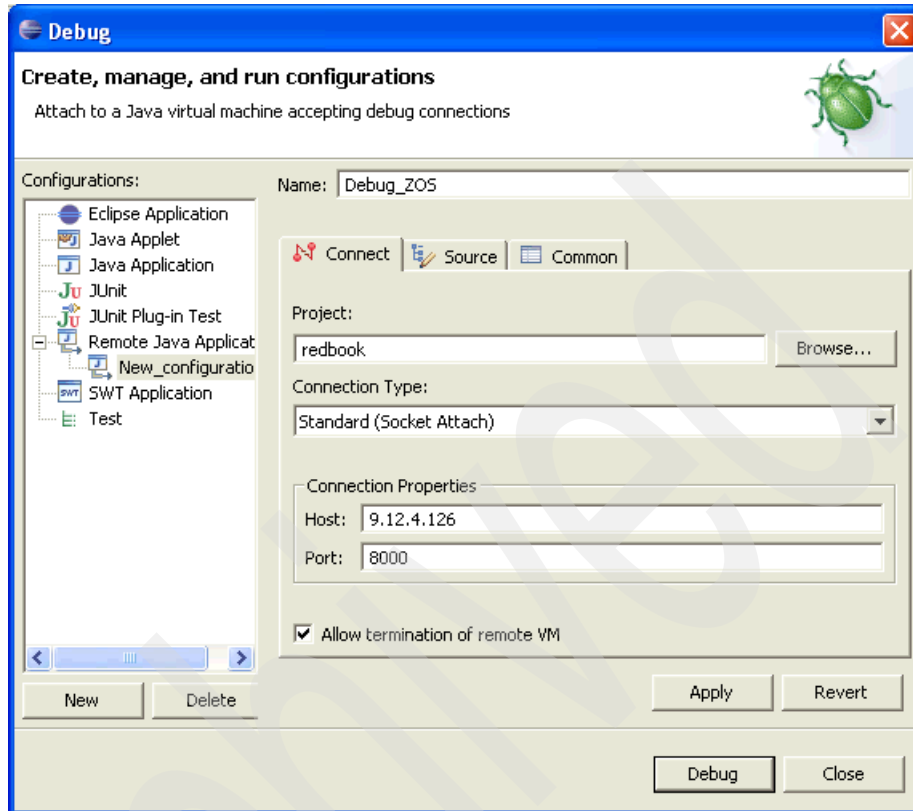


Figure 4-17 New remote debug

There are two properties in the Connection Properties section: Host and Port. We changed the Host value to 9.12.4.126 and left the Port value as 8000. Use appropriate values for your specific test situation.

It might be necessary to terminate the remote JVM from the debugger. For this purpose, click the Allow termination of remote VM check box.

The configuration window has two more tabs: Source and Common. The Source tab tells the debugger where to look for the source code of the Java program under debugging on the remote JVM. The Common tab controls other common configuration settings for remote debugging. We used default values under both tabs to simplify the demonstration.

Click the **Apply** button. The New_configuration under the Remote Java Application in the left pane changes to Debug_ZOS. The new debug configuration is ready to go.

Before starting the debugger, set up a breakpoint in ProcessTrans.java. Later on, the debugger locates the source code. Both source code and the breakpoint show up in the source code view of the debug perspective (Figure 4-18).

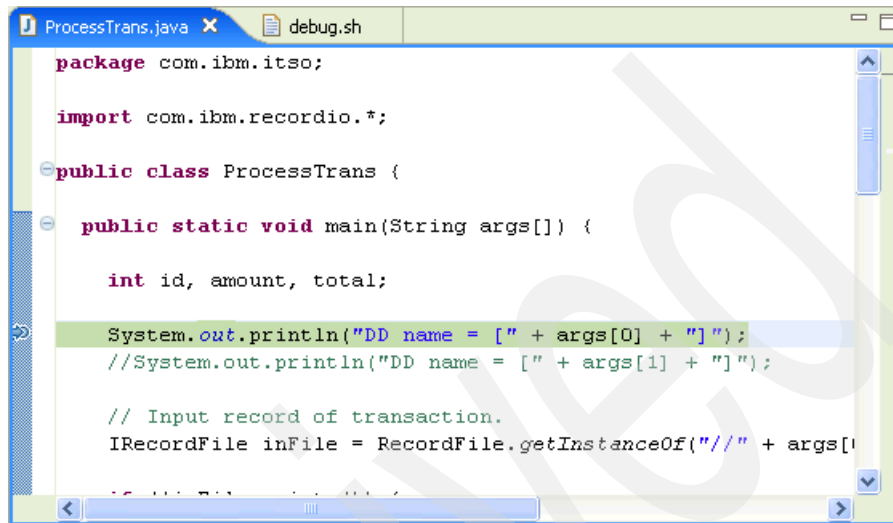


Figure 4-18 Source view of debug perspective

Click **Debug**. The launch attempts to connect to the JVM at the specified address and port. The result is displayed in the Debug view (Figure 4-19).

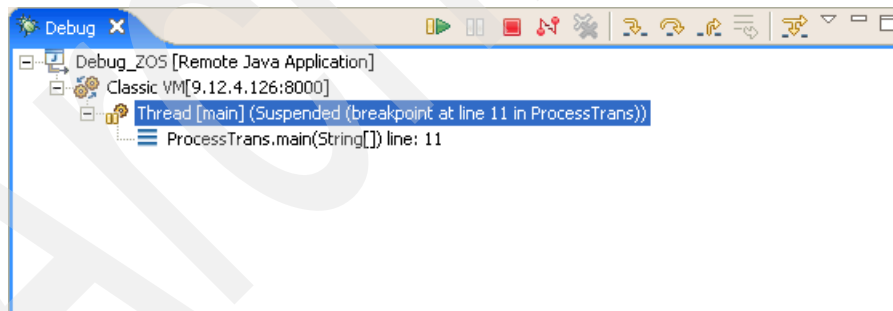


Figure 4-19 Debug view in debug perspective

From the Debug view, you can tell that the debugger connects to the JVM at 9.12.4.126:8000, and the main thread is suspended because of the breakpoint you set at line 11 in ProcessTrans.java. Using the icons in the Debug view, or shortcut keys, you can easily step through the program just as if it is running on your local machine.

The debugger can be disconnected, terminated, and relaunched from the context menu accessed by right-clicking the Debug view. Both disconnect and terminate will disconnect the debugger from the remote JVM. The difference is that disconnect does not terminate the remote JVM, but terminate does (if the Allow termination of remote VM is checked when configuration is done). Before the debugger can be relaunched, the remote JVM has to be started first. Currently, the remote JVM can only be started manually.

During the debugging process you can watch and inspect variable values using the Variables view (Figure 4-20). The arguments passed to the Java program are shown in this view. Right-click the Variables view to access a context menu from which you can manipulate the variables.

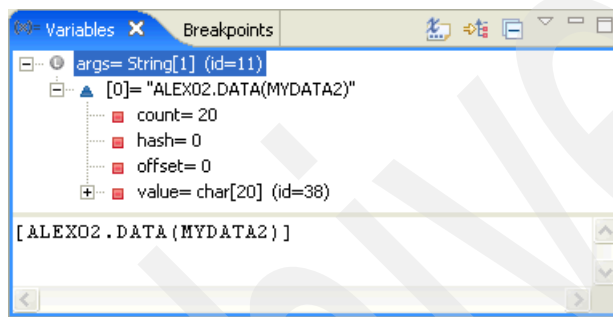



Figure 4-20 Variables view of Debug perspective

Add a new breakpoint to line 46 in the program. Then click the  button to resume running the program. The program run stops at the new breakpoint. Resume the program run again. The program stops at the same breakpoint because the breakpoint is in a loop. Resume the program run multiple times, then check the debug perspective.

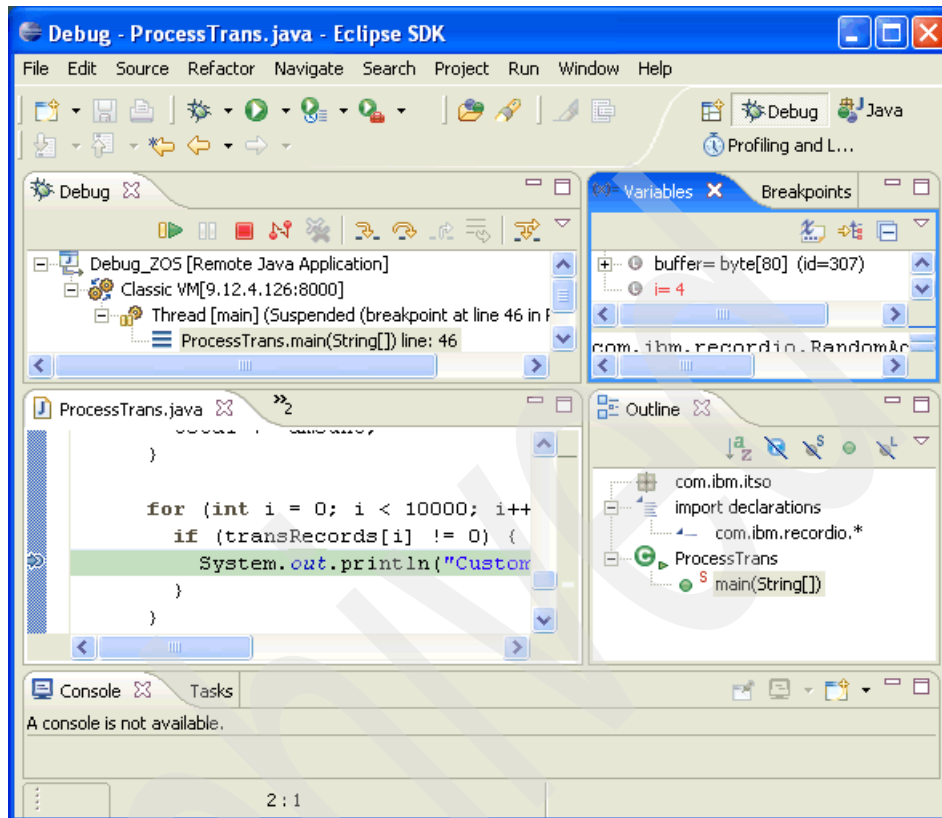


Figure 4-21 Debug perspective with breakpoints

Figure 4-21 shows the new debug perspective. The program has finished reading data from PDS member ALEX02.DATA(MYDATA2) and is about to print the data out to stdout using Java's **System.out.println()**.

Unfortunately, you cannot see the output of **System.out.println()** in the Eclipse workbench console. If the Java program is launched in USS, the output will be on your USS session screen. If the Java program is launched using BPXBATCH, you can find it either in a USS file or in SYSOUT, depending on the JCL/Proc.

Figure 4-22 on page 89 shows the USS session screen for our simple Java program debugging.

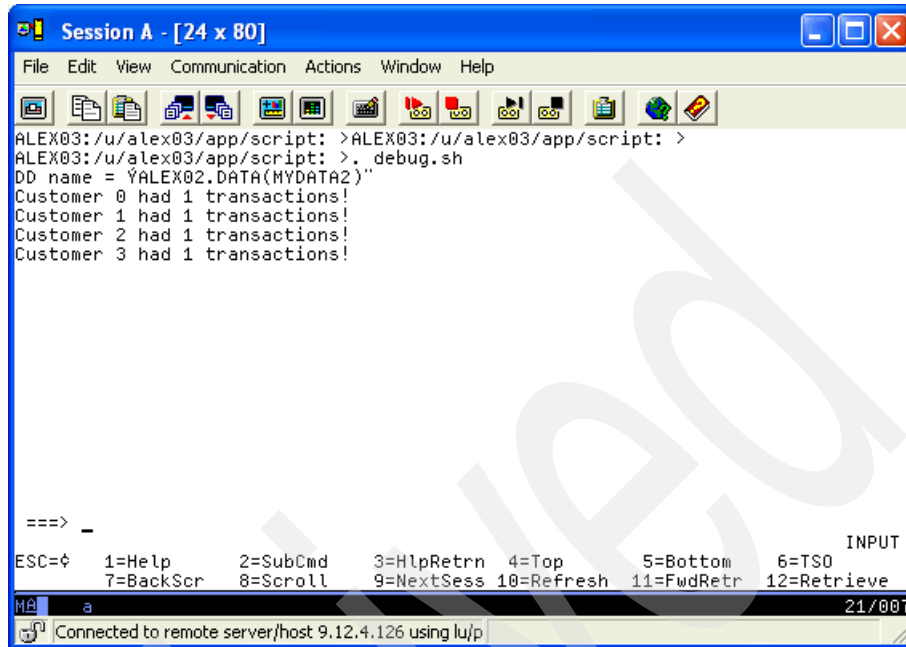


Figure 4-22 USS session screen for debugging

The debugger in the Eclipse Workbench is very powerful. It has far more functions than just the ones mentioned here. You can use almost all the functions it provides to debug a Java program running on z/OS. For more information about the Eclipse debugger, refer to the Eclipse Web site.

4.3.6 Local debugging

If the Java application being developed accesses only HFS files, DB2 for z/OS and WebSphere MQ for z/OS, it can be debugged locally in the Eclipse workbench.

Local files can be used for program debugging instead of HFS files. A JDBC Type 4 driver or DB2 Connect™ can be set up to facilitate the remote DB2 access. A WebSphere MQ client can be set up to access the queues in an MQ queue manager on z/OS.

Local debugging is more natural than remote debugging. There is no need to manually start a JVM before starting the debugger in the Eclipse workbench. The Eclipse workbench console handles standard I/O.

Refer to the previous section for a discussion of the basic use of the debugger.

4.3.7 Rational Application Developer considerations

Since *Rational® Application Developer* (RAD) is built on top of Eclipse, virtually all the previous discussion applies to RAD. You should be able to debug a Java program running on z/OS in practically the same way (if not exactly the same way) as that which we have already described.

Measuring performance and application profiling

This chapter describes some techniques and issue analysis methods for measuring performance in stand-alone and batch Java applications on z/OS.

A large part of this chapter is dedicated to accurate analysis using Test and Performance Tools Platform (TPTP). There are other profiling tools available, but we find TPTP to be one of the most useful ones.

The information provided in this chapter is of interest to those who have to deal with Java performance on platforms other than z, and with Java applications of types other than batch or stand-alone.

5.1 Performance analysis

This section provides basic information about how to do performance analysis. It includes code samples and a discussion of some useful analysis tools.

We describe some of the common performance problems that you can expect to encounter with a batch Java application and propose strategies for analysis that quickly identify the major contributors to a problem. The factors that constitute *application performance* have stayed constant over many years: CPU consumption, I/O, memory utilization, and middleware and operating system interactions. However, the relative contribution of each factor to overall performance has changed since the introduction of Java. This ratio of contribution continues to change as Java development practices, standards, and technologies progress.

Performance analysis is in essence the pursuit of awareness and knowledge. The more we understand the behavior of an application, the better we are able to apportion blame for poor performance to its components, and take action to improve the situation. The solution for a well understood problem is mostly self-evident, so the question is: do we understand the problem well enough. Often, we unfortunately do not.

5.1.1 Tooling overview

Probably the most important factor leading to a thorough understanding of a performance problem is the tools available to allow you to analyze the issue. For Java applications There are several categories of tools available for analyzing Java application performance. The tool categories are:

System	Tools that report the general health of the system on which application is running on, resource utilization, and any error conditions.
Load and stress test	This family of tools allows an application or transaction to be driven at a predefined load. They help expose performance issues as well as measuring success against a certain operation performance goal.
Profiling	These tools can accurately record the execution of a Java program by recording every method call, thereby enabling low-level analysis of the actual Java code being run.
J2EE monitors	These tools take a higher level view, often in production, of the performance and interaction of the major components within a J2EE application such as EJBs, JSPs, and external middleware resources.

Middleware

Specific tools and utilities that give a view of the health and performance of middleware services on which a Java application might rely.

5.1.2 Establishing a goal

Part of the awareness that is key to identifying and solving performance problems is establishing a goal that the application must meet. In many cases this goal may be broadly defined as a response time target, for example, as part of a Service Level Agreement (SLA) with the end user or customer. These response time goals rarely, if ever, contain an element of resource utilization or cost. Additional targets in an SLA may refer to a number of users, which must go through a translation, often some form of usage scenario or stress test tool script, into the more useful metric of Transactions Per Second (TPS). A comprehensive performance goal for an interactive or online Java application would include, or allow to be calculated, the following:

- ▶ Number of users
- ▶ Response time
- ▶ Resource utilization
- ▶ Transactions per second
- ▶ Operational cost

In a batch Java application the goal would be slightly different due to its iterative nature and lack of interactive users. In this case, performance goals might include, for example:

- ▶ Number of Iterations®
- ▶ Number of reads and writes per iteration
- ▶ Elapsed execution time
- ▶ Resource utilization
- ▶ Operational cost

These goals can then be either compared directly against a recently developed Java application, or ideally be used to help govern the design and development of a new application from the beginning. Defining the target performance accurately improves analysis efficiency and allows a better understanding of the improvements required.

Here is an example target for a Java batch application:

A key file of 2 million records driving a batch process, in which each iteration requires two additional reads and one update, is required to complete within

60 minutes, and there will be approximately 250 MIPS of capacity available at that time.

Two million iterations, each containing 2 reads, 1 update and any business logic in 60 minutes, or 3600 seconds. In this 3600 second period 250 million instructions (MI) is available each second, for a total of 270000 MI.

$(250 \text{ MI} \times 3600) / 2000000 = 270000/2000000 \text{ MI}$

This resolves to 0.45 MI per iteration.

This is clearly a very approximate calculation, but it would be accurate enough to guide development or to assist in initial performance analysis and tuning.

5.2 Overview of analysis strategies

If a Java batch program is perceived to have poor performance through observation of its resource utilization or elapsed execution time, then the first logical step would be to analyze the Java program itself. There are several analysis strategies and techniques that provide differing benefits. These strategies are discussed in the following sections.

5.2.1 Statistical

This approach involves driving the application with a defined and repeatable workload and observing its resource utilization, and, in the case of batch, its elapsed execution time. It is normally used to determine if there is a performance problem rather than to help in analysis, but if broad system and environment changes are made between runs, this technique can be used to measure the improvement over a previously established baseline. Such changes could be, for example, tuning the Garbage Collection (GC) policy or changing the configuration of middleware resources. Statistical analysis can mostly be carried out without the need for in-depth Java or application knowledge.

5.2.2 Fractional

This is a simple technique, but it can require code changes. It splits up the major architectural components of an application and measures each one, as well as the entire transaction or iteration in batch. It can be as simple as printing to the logs millisecond timings regarding certain method calls, or it can involve more complex system calls to retrieve real resource utilization. Because the overhead can be very low it allows the application to be run at nearly full execution speed, which can help expose some of the more illusive locking and timing problems.

5.2.3 Elimination

Elimination or stubbing can be very useful in cases where an application relies on middleware or communication with other systems. It involves removing calls to certain functions and replacing them with a zero cost functional equivalent. This technique often requires more code changes than fractional analysis because in most cases the return data from the eliminated function will have to be simulated. Its benefit is that it can help work out whether there is a base performance problem with the application code or the observed issue is more a result of long and variable middleware or communication response times.

5.2.4 Accurate

This strategy involves profiling the application in great detail. However, this cannot be done at anywhere near full execution speed. This means that some of the more subtle issues that may present themselves at higher transaction rates may not be visible, but on the other hand you will be able to gain a deeper understanding of the execution flow of vendor and third-party code as well as your own.

5.3 Statistical analysis

As mentioned previously, this technique mainly is used informally to establish the presence of a performance problem, but it can also be used to analyze the issue and test improvements. It involves:

- ▶ Choosing a repeatable load of sufficient size
- ▶ Executing the Java program with this load
- ▶ Measuring the resource utilization during execution
- ▶ Changing environmental settings or configuration and re-testing

In this example we analyze a batch Java program, used during the writing of this book, that created and populated several tables in a test database. It loads several flat files from the HFS that contain example names and addresses and inserts randomized versions of them into two tables: *Customer* and *Account*. For the most part it inserts one row into the Customer table, then one row into the Account table; occasionally it may randomly choose to insert two or three account rows per customer.

Twelve tests were run with six different data volumes and two different JDBC drivers (the network-based Type 4 driver and the local Type 2 driver). Due to the randomization described previously, the number of rows inserted into the Account database differed between the runs but only by a small amount.

Therefore this can effectively be eliminated as a factor. Figure 5-1 shows the results of the test runs. You can clearly see that using the JDBC Type 2 driver reduces CPU consumption and elapsed time and also increases CPU utilization efficiency over the Type 4 driver. This example is admittedly simple, but the technique, when applied formally, can yield a deeper understanding of the performance envelope of an application under differing environmental conditions. In addition to collecting elapsed time and CPU consumption, other external measurements can be taken, including:

- ▶ Disk I/O
- ▶ Network I/O
- ▶ Garbage collection statistics

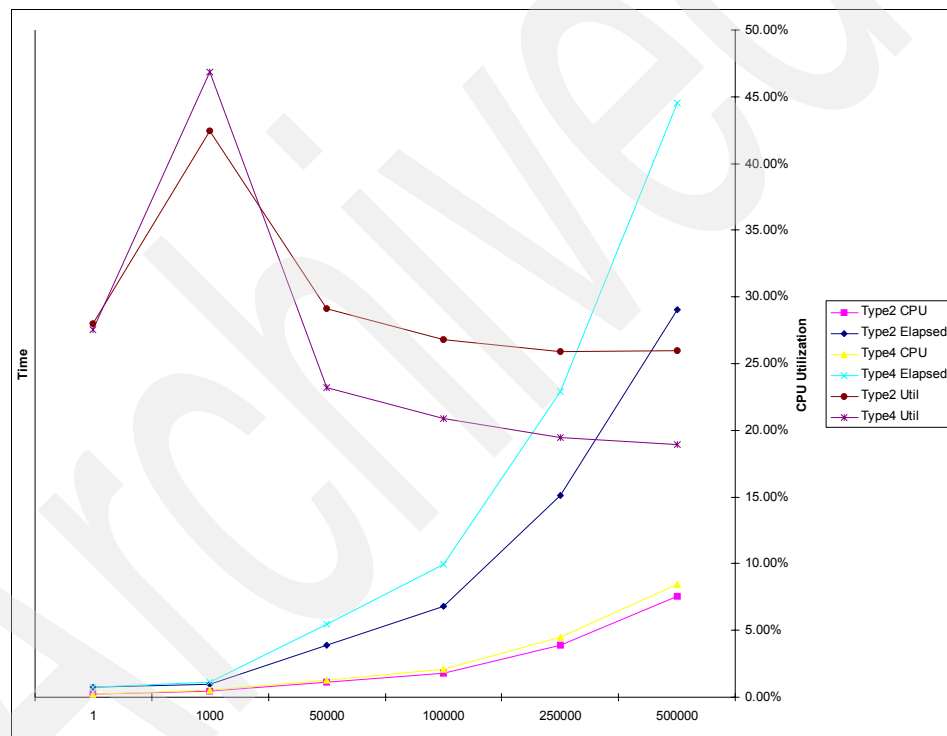


Figure 5-1 Statistical analysis of CreateDatabase

5.4 Fractional analysis

Using fractional analysis you can apportion relative blame for poor performance to different sections of a program by surrounding calls to architecturally significant areas of code with timing calls that report the cost of that function. In

its simplest and most often used form, this involves calling the **System.currentTimeMillis()**, which returns the current system time in milliseconds, before and after the call. Subtracting the *start* time from the *stop* time yields the number of milliseconds the timed call took to execute (Example 5-1).

Example 5-1 Simple fractional analysis using System.currentTimeMillis()

```
public class DatabaseUpdate(DataObject dataObject) {  
  
    .....  
  
    long startTime = System.currentTimeMillis();  
  
    dataObject.store();  
  
    long stopTime = System.currentTimeMillis();  
  
    System.out.println("Time to store =" + (stopTime - startTime) + "ms");  
  
    .....  
  
}
```

This simple form of the technique can be extremely useful, but it reports “wallclock” timing rather than real CPU time and by itself would not be able to differentiate between a function that consumes a large amount of CPU and another one which waits for some form of external resource. In addition, on a shared system such as zSeries, it is very possible that a process would be suspended by the operating system in the middle of a timed call, during which time the clock returned by **currentTimeMillis** keeps on running.

This variability can be mitigated to some extent by collating the reported timings and calculating the minimum, maximum, and average times. If the code path is the same on every timed call, then over many calls the minimum time may represent the best time achievable and may help eliminate OS and middleware variability.

At some point, when a large number of functions are being timed many times over, then analyzing this data in the log files may become difficult. A slightly more sophisticated timing mechanism could be created that collects statistics during execution but only reports on completion, as in Example 5-2 on page 114.

Example 5-2 A more sophisticated System.currentTimeMillis() timing class

```
package com.ibm.itso.timer;

import java.io.PrintStream;
import java.util.Enumeration;
import java.util.Hashtable;

public class Timer {

    private static Hashtable timers = new Hashtable();
    private long start = 0;
    private long stop = 0;
    private long invocations = 0;
    private long total = 0;
    private long average = 0;
    private long minimum = Integer.MAX_VALUE;
    private long maximum = 0;
    private String name = null;;

    public static Timer getTimer(String timerName) {
        Timer timer = (Timer) timers.get(timerName);
        if (timer == null) {
            timer = new Timer(timerName);
            timers.put(timerName, timer);
        }
        return timer;
    }

    public Timer(String name) {
        this.name = name;
    }

    public void start() {
        this.start = System.currentTimeMillis();
    }

    public void stop() {
        stop = System.currentTimeMillis();
        invocations++;
        long time = stop - start;
        total += time;
        if (time < minimum) {
            minimum = time;
        }

        if (time > maximum) {
            maximum = time;
        }
    }
}
```

```

    }

    public static void reportAll(PrintStream printStream) {
        Enumeration timerList = timers.elements();
        while (timerList.hasMoreElements()) {
            ((Timer) timerList.nextElement()).report(printStream);
            printStream.println("");
        }
    }

    public void report(PrintStream printStream) {
        if(invocations>0) {
            average = total / invocations;
        }
        printStream.println("Timer: " + name);
        printStream.println("  Invocations: " + invocations);
        printStream.println("  Total Time: " + total);
        printStream.println("  Average Time: " + average);
        printStream.println("  Minimum Time: " + minimum);
        printStream.println("  Maximum Time: " + maximum);
    }
}

```

The use of this timer consists of a call to the static method **Timer.getTimer()** passing the name of the timer you wish to create and retrieve, followed by a call to start on that particular timer before the method call you wish to time. After the call of interest returns, the stop method on the timer is called. An example is shown in Example 5-3.

Example 5-3 Usage of the Timer class

```

public class DatabaseUpdate(DataObject dataObject) {

    ....

    Timer myTimer = Timer.getTimer("DataObject.store()");
    myTimer.start();

    dataObject.store();

    Timer.getTimer("DataObject.store()").stop();

    ....

}

```

At the end of the execution of the program, you would then ask the timer to report the totals for an individual timer or all timers passing the `PrintStream` that the statistics should be written to.

Example 5-4 Reporting statistics from the Timer class

```
public static void main(String args[]) {  
  
    Timer programTimer = Timer.getTimer("ProgramTimer");  
    programTimer.start();  
  
    .....  
  
    programTimer.stop();  
  
    Timer.reportAll(System.out);  
}
```

This would print output in the format shown in Example 5-5.

Example 5-5 Output from the Timer class

```
Timer: ProgramTimer  
  Invocations: 1  
  Total Time: 4599  
  Average Time:4599  
  Minimum Time:4599  
  Maximum Time:4599  
  
Timer: DataObject.store();  
  Invocations: 50  
  Total Time: 3003  
  Average Time:61  
  Minimum Time:50  
  Maximum Time:82
```

This example of the `Timer` class can be useful, but it is not thread safe because many real world applications, including Java batch, would utilize multiple threads simultaneously. A more thread-safe version of the timer would have to allow all threads to be able to start and stop the same `Timer` but the overall `Timer` statistics to be collated centrally. The sample code shown in Example 5-6 on page 101 enables this behavior.

Example 5-6 Example of a thread safe Timer class

```
package com.ibm.itso.timer;

import java.io.PrintStream;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Stack;

public class ThreadSafeTimer {

    private static Hashtable timers = new Hashtable();

    private static ThreadlocalTimers threadlocalTimers = new
ThreadlocalTimers();

    private long invocations = 0;
    private long total = 0;
    private long minimum = Integer.MAX_VALUE;
    private long maximum = 0;
    private String name = null;

    private static class ThreadlocalTimers extends ThreadLocal {

        public Object initialValue() {
            return new Hashtable();
        }

        public Stack getStack(String timerName) {
            Hashtable stacks = (Hashtable)super.get();
            Stack stack = (Stack) stacks.get(timerName);
            if(stack==null) {
                stack = new Stack();
                stacks.put(timerName, stack);
            }
            return stack;
        }
    }

    private class Timelet {
        long start = 0;

        Timelet(long start) {
            this.start = start;
        }
    }

    public static synchronized ThreadSafeTimer getTimer(String timerName) {
        ThreadSafeTimer timer = (ThreadSafeTimer) timers.get(timerName);
```

```

        if (timer == null) {
            timer = new ThreadSafeTimer(timerName);
            timers.put(timerName, timer);
        }
        return timer;
    }

    public ThreadSafeTimer(String name) {
        this.name = name;
    }

    public void start() {
        Stack timeletStack = threadlocalTimers.getStack(this.name);
        timeletStack.push(new Timelet(System.currentTimeMillis()));
    }

    public synchronized void stop() {
        Stack timeletStack = threadlocalTimers.getStack(this.name);
        long start = ((Timelet) timeletStack.pop()).start;
        long stop = System.currentTimeMillis();
        long time = stop - start;
        invocations++;
        total += time;
        if (time < minimum) {
            minimum = time;
        }
        if (time > maximum) {
            maximum = time;
        }
    }

    public static void reportAll(PrintStream printStream) {
        Enumeration timerList = timers.elements();
        while (timerList.hasMoreElements()) {
            ((ThreadSafeTimer) timerList.nextElement()).report(printStream);
            printStream.println("====");
        }
    }

    public void report(PrintStream printStream) {
        long average = 0;
        if (invocations > 0) {
            average = total / invocations;
        }

        printStream.println("Timer: " + name);
        printStream.println(" Invocations: " + invocations);
        printStream.println(" Total Time: " + total);
        printStream.println(" Average Time: " + average);
        printStream.println(" Minimum Time: " + minimum);
    }

```

```

        printStream.println(" Maximum Time: " + maximum);
    }
}

```

This thread-safe version of the timer has certain advantages and disadvantages. Each thread has a stack for each named timer, allowing timer calls to be nested, within recursive methods, for example. In addition, the application does not have to worry about keeping a reference to a particular Timer in scope for it to be stopped, because the static call to `Timer.getTimer("MyTimer").stop()` will pop the “timelet” most recently started in this thread off the stack, stop it, and add the figure to the aggregate statistics. However, this means that calls to the stop method of a certain timer will follow the reverse order in which they were started. In addition, this example does not calculate its own overhead.

5.4.1 TIMEUSED

Millisecond timings help you to a point, but their variability and relatively low resolution, if you consider the speed of modern processors, are limiting factors. Ideally, you would want to report the CPU consumption as the operating system’s scheduler “sees” it. Fortunately, on z/OS you have such a function: the assembler TIMEUSED macro.

Mike Cox from the IBM Washington Systems Center was first to wrap this existing macro in JNI ‘c’ and Java, thus making it available to any Java application. A timing class was then built around it for ease of use.

A call to the TIMEUSED function returns the total CPU time consumed by the TCB, or thread, at that particular moment in time, in microseconds. As with `System.currentTimeMillis()`, you would take a measurement before the function of interest and another after it, and then subtract one from the other, which produces an accurate figure of CPU time that the function consumed.

Because you are now dealing with a counter that has 1000 times better resolution than the millisecond timer we used before, it allows you to time much smaller events. It also means that the overhead of the timing calls themselves would be visible in the reported time of a very small method. You need to think about a calibration function that calculates this overhead, allowing it to be removed again.

Our approach is to extend Mike’s original concept and merge it with the thread-safe code. The general structure of this new code is an assembler route that makes the call to the TIMEUSED macro. This is called from a small c wrapper, which in turn is called via JNI from our ThreadSafeTimeUsed class.

Note: To compile the following code you need to have a z/OS C/C++ compiler installed and configured.

Create a directory on z/OS called *timeused* and save the assembler code shown in Example 5-7 in a file called *cpu.s* in this new directory.

Example 5-7 cpu.s, the assembler containing the call to the TIMEUSED macro

```
*      PRINT NOGEN
R1     EQU  1
R2     EQU  2
R3     EQU  3
R4     EQU  4
R5     EQU  5
R6     EQU  6
R7     EQU  7
R8     EQU  8
R9     EQU  9
R10    EQU 10
R11    EQU 11
R12    EQU 12
R13    EQU 13
R14    EQU 14
R15    EQU 15
CPU_PARMS DSECT
CPU_PARMS_START DS OF
CPU_PARMS_FIRST_PTR DS F
CPU      RMODE ANY
CPU      AMODE 31
CPU      CSECT
        USING CPU,R15
        B     DOIT
        DC    CL8'CPU--STR'
DOIT     DS    0H
        SAVE  (14,12)
        LR    R10,R15
HERE     DS    0H
        L     R2,0(,R1)
        TIMEUSED STORADR=(2),LINKAGE=SYSTEM,CPU=MIC
        RETURN (14,12),RC=0
        DC    CL8'CPU--END'
        END
```

Next, within the *timeused* directory create the subdirectory structure *com/ibm/itso/timer*. This can be achieved with a single command, as shown in Example 5-8 on page 105.

Example 5-8 Making the java package directory structure

```
cd /u/user/timeused
mkdir -p com/ibm/itso/timer
```

Note: To accelerate the following procedure, you can use the finished ThreadSafeTimeUsed class from the code listing at the end of this chapter instead of copying and adjusting the ThreadSafeTimer code provided previously.

Next, take the ThreadSafeTimer code and place it in the file `./timeused/com/ibm/itso/timer/ThreadSafeTimeUsed.java`. Edit this file, change the class name to ThreadSafeTimeUsed, and add the native method call, as shown in Example 5-9.

Example 5-9 Changing ThreadSafeTimer into ThreadSafeTimeUsed

```
.....
public class ThreadSafeTimeUsed {
.....

public static native long timeUsed();
.....
```

You can compile this class in the regular manner and then use the javap command line tool, which is located in the JVMs bin directory, to generate a header file that creates a c header file containing the function signature that you will have to use (Example 5-10).

Example 5-10 Compiling ThreadSafeTimeUsed and running javap over it

```
cd /u/user/timeused
javac com/ibm/itso/ThreadSafeTimeUsed.java
javap -jni com.ibm.itso.ThreadSafeTimeUsed
```

This creates the file `com_ibm_itso_timer_ThreadSafeTimeUsed.h` in the `./timeused` directory, as shown in Example 5-11.

Example 5-11 com_ibm_itso_timer_ThreadSafeTimeUsed.h

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

#ifdef _Included_com_ibm_itso_timer_ThreadSafeTimeUsed
#define _Included_com_ibm_itso_timer_ThreadSafeTimeUsed
```

```

#ifdef __cplusplus
extern "C" {
#endif
/* Inaccessible static: timers */
/* Inaccessible static: threadlocalTimers */
/*
 * Class:      com_ibm_itso_timer_ThreadSafeTimeUsed
 * Method:     timeUsed
 * Signature:  ()J
 */
JNIEXPORT jlong JNICALL Java_com_ibm_itso_timer_ThreadSafeTimeUsed_timeUsed
    (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif

```

Next, in the same directory, create the c wrapper called `com_ibm_itso_timer_ThreadSafeTimeUsed.c` that will bridge between Java and the assembler function (Example 5-12).

Example 5-12 com_ibm_itso_timer_ThreadSafeTimeUsed.c

```

#include "com_ibm_itso_timer_ThreadSafeTimeUsed.h"

void CPU (void *);
#pragma linkage(CPU,OS)

JNIEXPORT jlong JNICALL Java_com_ibm_itso_timer_ThreadSafeTimeUsed_timeUsed
    (JNIEnv * env, jobject thisObject) {
    jlong microseconds = 0L;

    CPU(&microseconds);

    return microseconds;
}

```

Finally, compile and link the c and assembler code (Example 5-13) into a shared object, a .so file, which you dynamically load in your class via a static call to `System.loadLibrary`.

Example 5-13 Compilation of the native parts of ThreadSafeTimeUsed

```

c89 -o libtimeused.so -W "l,xplink,dll" -W
"c,langlvl(extended),xplink,dll,exportall" -I/usr/lpp/java/J1.4/include

```

```
com_ibm_itso_timer_ThreadSafeTimeUsed.c cpu.s  
/usr/lpp/java/J1.4/bin/classic/libjvm.x
```

The parts of the compile/link command shown in Example 5-13 have the following meanings:

c89	The command to invoke the c compiler.
-o libtimeused.so	This is the output file, the shared library that you wish to build.
-W "l,	Pass the following options -W to the link l phase of the compiler.
xplink,dll"	Tells the linker that you wish to create a shared library dll, and it is to use high performance linkage xplink.
-W "c,	Pass the following options -W to the compile c phase of the compiler.
langlvl(extended),	Enable language extensions used by the JNI header files such as the 'long long'.
xplink,dll,	As previously described.
exportall	Make call functions in the sll callable by external programs.
-I/usr/lpp/java/J1.4/include	Look -I in the /usr/lpp/java/J1.4/include directory, in addition to the default directories, for header files, in this case JNI header files.
com_ibm_itso_timer_ThreadSafeTimeUsed.c	Your c program.
cpu.s	Your assembler function.
/usr/lpp/java/J1.4/bin/classic/libjvm.x	A "side deck" file that defines function entry points to the core Java shared library (libjvm.so).

This command compiles your c code, then your assembler code, then it links and binds them together into a shared library that can be loaded and called dynamically from java.

To load this library the static code block shown in Example 5-14 is added to the top of the class.

Example 5-14 Statically loading the timeused.so library

```
static {
    System.loadLibrary("timeused");
}
```

In your start and stop method, change **System.currentTimeMillis** to a call to the native routine **timeUsed()**, which returns a 64-bit long value containing the CPU consumed by the thread at that instant.

You also need to add the simple calibration routine shown in Example 5-15 because your resolution is now much higher and the overhead of the calls to your timing code may well render smaller timed events inaccurate.

Example 5-15 Calibration method

```
static {
    ThreadSafeTimeUsed calibration = null;
    for (int i = 0; i < 5; i++) {
        calibration = ThreadSafeTimeUsed
            .getTimer("*Internal Calibration Timer " + i);
        for (int j = 0; j < 1000; j++) {
            calibration.start();
            calibration.stop();
        }
        long localOverhead = calibration.total / calibration.invocations;
        if (localOverhead < overhead) {
            overhead = localOverhead;
        }
        timers.remove("*Internal Calibration Timer " + i);
    }
}
```

The calibration routine attempts to call the start and stop methods enough times to invoke JIT compilation, and over 5 runs of 1000 invocations of start and stop it takes the lowest average time.

It is important to remember that the reported times are guides and will help determine if a certain section of code is a major issue or not, but a single invocation of a timed method is not enough to establish a fair or accurate figure. Many invocations should be repeatedly timed in the same JVM instance to help eliminate JIT compilation garbage collection and any variable system overhead.

To use the ThreadSafeTimeUsed function you first need to make sure that the libtimeused.so library can be located by adding the directory it is in to the LIBPATH environment variable. This can be done with the command shown in Example 5-16.

Example 5-16 Adjusting the LIBPATH environment variable

```
export LIBPATH=LIBPATH:/u/user/timeused
```

What follows is an example of the use of the ThreadSafeTimeUsed class. It is timing several elements of a simple interest rate calculation program.

Example 5-17 Example usage of ThreadSafeTimeUsed

```
public class InterestCalculation {

    public static void main(String[] args) {

        BigDecimal fivePercent = new BigDecimal("1.05");

        ThreadSafeTimeUsed.getTimer("Whole Program").start();

        for(int i=1;i<1000;i++) {

            ThreadSafeTimeUsed.getTimer("Customer.findByID").start();
            Customer customer = Customer.findByID(new Integer(i));
            ThreadSafeTimeUsed.getTimer("Customer.findByID").stop();

            if(customer != null) {

                ThreadSafeTimeUsed.getTimer("Customer.getAccounts").start();
                Enumeration accounts = customer.getAccounts().elements();
                ThreadSafeTimeUsed.getTimer("Customer.getAccounts").stop();

                ThreadSafeTimeUsed.getTimer("Accounts loop").start();
                while(accounts.hasMoreElements()) {

                    Account account = (Account)accounts.nextElement();
                    BigDecimal balance = account.getBalance();
                    balance.multiply(fivePercent);
                    account.setBalance(balance);

                    ThreadSafeTimeUsed.getTimer("Account.store").start();
                    account.store();
                    ThreadSafeTimeUsed.getTimer("Account.store").stop();

                }

                ThreadSafeTimeUsed.getTimer("Accounts loop").stop();

            }

        }

        ThreadSafeTimeUsed.getTimer("Whole Program").stop();
    }
}
```

```
ThreadSafeTimeUsed.reportAll(System.out);  
  
    }  
}
```

The test reported the timings shown in Example 5-18. Notice the high figures for the maximum times, which is probably due to JIT compilation, one of the invocations of the timed methods hit a JVM defined threshold and was then JIT compiled. This expensive compilation task was “charged” to this particular method invocation.

It would be trivial to change the code to eliminate the highest reported timings, and thus prevent JIT compilation overhead from skewing the results too much.

Example 5-18 Example output of ThreadSafeTimeUsed

```
Timer: Accounts loop  
  Invocations: 999  
  Total Time: 3336641  
  Average Time: 3339  
  Minimum Time: 1372  
  Maximum Time: 293959  
  Overhead: 2 (Calculated but not removed)  
  
Timer: Customer.findByID  
  Invocations: 999  
  Total Time: 3442869  
  Average Time: 3446  
  Minimum Time: 1660  
  Maximum Time: 443266  
  Overhead: 2 (Calculated but not removed)  
  
Timer: Whole Program  
  Invocations: 1  
  Total Time: 13504371  
  Average Time: 13504371  
  Minimum Time: 13504371  
  Maximum Time: 13504371  
  Overhead: 2 (Calculated but not removed)  
  
Timer: Customer.getAccounts  
  Invocations: 999  
  Total Time: 6700231  
  Average Time: 6706  
  Minimum Time: 3353  
  Maximum Time: 138433  
  Overhead: 2 (Calculated but not removed)
```

```
Timer: Account.store
  Invocations: 1185
  Total Time: 3306555
  Average Time: 2790
  Minimum Time: 1366
  Maximum Time: 292003
  Overhead: 2 (Calculated but not removed)
```

Attention: The TIMEUSED function is not eligible for zAAP. When in profiling or debug mode this may not be a problem, but when you use this function for logging purposes you should keep this in mind.

5.5 Elimination analysis

The following definitions are important for a complete understanding of this topic:

► **Internal Throughput Rate or ITR**

This is the number of units of work accomplished per unit of processor busy time in an unconstrained environment.

Internal Throughput Rate = Units of Work / Processor Time Consumed

► **External Throughput Rate or ETR**

This is the number of units of work accomplished per unit of elapsed time including the elapsed time of external resources.

External Throughput Rate = Units of Work / Elapsed Time

In many cases a Java application will make synchronous requests to external resources. Some of these calls may involve network communications out of the machine where the application is running. External calls, especially in test environments, can have extremely variable response times that will lead to different transaction times for the Java application making the calls. Therefore, it can be difficult to gain a true picture of the performance of an application when presented with this variability.

One could use broader statistical techniques to calculate the ETR and ITR of an application, but in many cases a deeper understanding of the performance of an application can be gained, and issues exposed, by driving it to its limits. This can be difficult if synchronous calls to external resources limit throughput.

A technique to establish the base performance of a transaction that could include variable external requests is to simply eliminate them by replacing the calls with methods that simulate the response from the original resource. For example, if

you are testing an application that calls a CICS transaction but the test CICS region being used responds slowly or inconsistently, the call could be eliminated or “stubbed.” This simulated CICS call would return immediately, allowing the Java transaction to continue.

This would make the determination of the performance of the Java application itself much more straightforward when using load and stress testing tools. Work could continue on tuning the Java application itself without having to constantly factor in the variable response time of the external resources.

In some cases the data returned by a call may be too complex to simulate manually, or dependant on the form or content of the request data. In these cases the response of a real request can be “recorded” and then played back in the simulated request. Even so, not all external requests may be able to be eliminated.

Clearly, true response times in production will include the external calls that have been eliminated in this testing technique. For this reason, elimination can only be used as a strategy to allow developers to concentrate on their own code’s performance. It can, however, be a very helpful practice.

5.6 Accurate analysis

Accurate analysis allows us to understand in detail the actual execution of the Java code in question. This is normally done using a profiling tool. Profiling tools can add a significant overhead in both CPU utilization and elapsed execution time. Some performance issues are only visible at full execution speed, such as contention or locking. However, application profiling allows the design and implementation of not just the self-written application code to be analyzed, but also vendor and open source code that is normally hidden from the application developer.

One of the most exciting profiling tools available today is the Test and Performance Tools Platform (TPTP), which is part of the Eclipse project. TPTP is an open source initiative that attempts to provide many of the testing and performance analysis functions required by Java professionals. We concentrate on the application profiling aspect of TPTP in this section.

5.6.1 Installing TPTP

At a high level, the steps to install TPTP on the client and System z server are the following:

1. Download and install Eclipse.

2. Download and install TPTP and its requirements.
3. Download, install and configure the Remote Agent Controller for z/OS.

Detailed instructions for these steps are included in the following sections.

Downloading and installing Eclipse

For application profiling we recommend a client machine configuration with a greater than 2Ghz processor and 1GB or more of memory. It is possible to profile on a lower specification machine, but the size and detail of the profile information that can be visualized may be lower.

Download Eclipse from the following Web site:

<http://www.eclipse.org/downloads>

Select the latest released version for your operating system. At the time of writing this was 3.1.2 for our Windows operating system.

Note If you will be running Eclipse on Linux® then you have 2 choices: a version of Eclipse that uses Motif as its underlying Window toolkit for SWT, or a version that uses GTK2. The Motif implementation is generally considered to be faster, but GTK2 delivers a more visually pleasing experience.

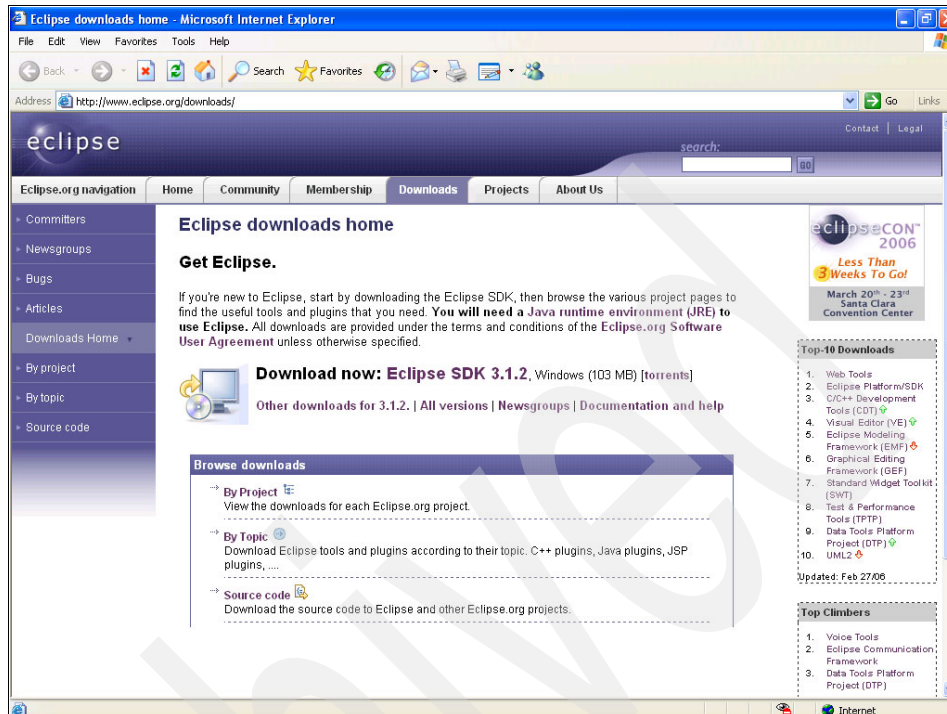


Figure 5-2 The Eclipse.org downloads page

When requested to, select a download mirror that is closest to you and download the Eclipse install zip file anywhere on disk. The Eclipse install is very simple and straightforward. The zip file consists of an Eclipse directory that is a self contained instance of the software. For example, unzipping it to the root of C:\ creates c:\eclipse directory, in which the entire installation resides.

Eclipse requires a working JDK™ or JRE installation on the target workstation, but it does not come with one. When Eclipse starts up for the first time it will attempt to detect a suitable JDK or JRE to use, a setting that can be changed later on if required.

The Eclipse download page has a link to some available JREs that you can use if your workstation does not have a JDK or JRE installed. Alternatively, you could visit:

<http://www.ibm.com/developerworks/java>

Open the newly created eclipse folder and start the workbench by double-clicking the Eclipse program icon (Figure 5-3 on page 115). At a later stage you can create a link to this program on your desktop or as a menu item.

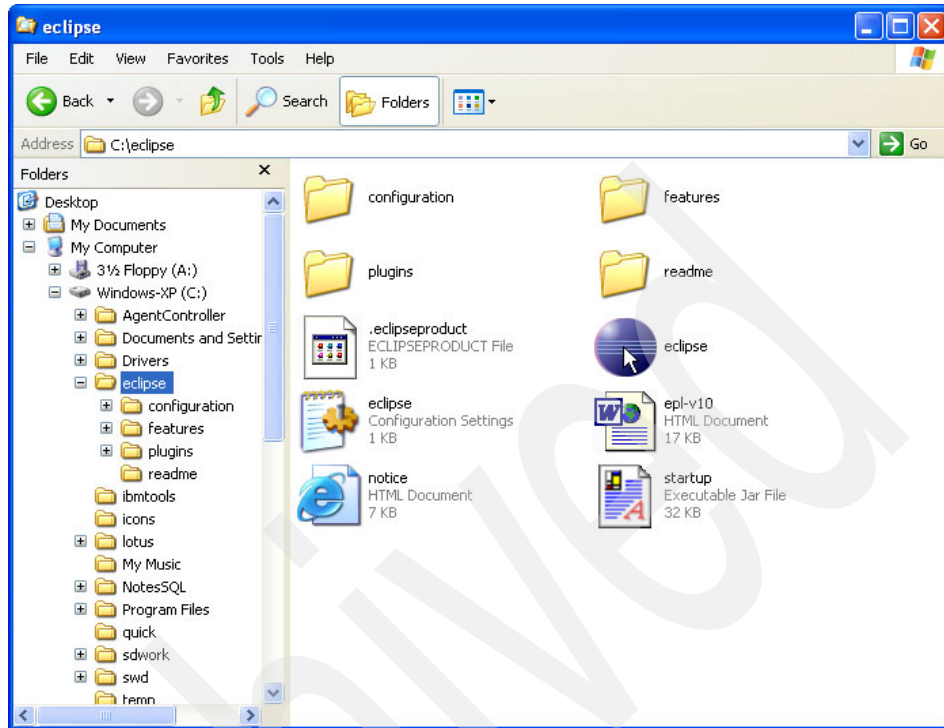


Figure 5-3 Starting Eclipse

The Eclipse workbench should now start and present the default welcome page. You can now close Eclipse and move on to installing TPTP.

TPTP has two Eclipse plug-in dependencies that you have to install before you continue. These are the Eclipse Modeling Framework (EMF) and the XML Schema Infoset Model (XSD). There are two ways to download and install TPTP and these dependencies:

- ▶ Manually download each component as a zip file and extract it to the eclipse directory.
- ▶ Use the Automatic update and Install feature built into the Eclipse workbench to download and install the features.

Manually installing TPTP

To download TPTP and the two additional components go to the following Web site:

<http://www.eclipse.org/tptp>

Click the **Downloads** link. At the bottom of the page you should see links to the download pages for the additional software requirements for TPTP. At the time of this writing the dependencies were EMF version 2.1.2 and XSD version 2.1.2. Follow these links and download the associated zip files, selecting your local file mirror as required.

Return to the TPTP Project Download page and download the single zip file containing the current TPTP Runtime for all project sub components, selecting your local file mirror as required. You will now have three zip files on your workstation, for example:

1. emf-sdo-SDK-2.1.2.zip
2. xsd-SDK-2.1.2.zip
3. tptp.runtime-TPTP-4.1.0.1.zip

With the Eclipse workbench shut down, unzip each of the zip files into the directory that contains the eclipse directory created previously. For example, if Eclipse is in the C:\eclipse directory, uncompress the zip files into the C:\ directory.

Automated installation

Eclipse has the capability to automatically install new components or update existing components over the Internet. To install TPTP and its dependencies using this mechanism follow these steps:

1. Start Eclipse as outlined previously. The main welcome page is displayed (Figure 5-4 on page 117).

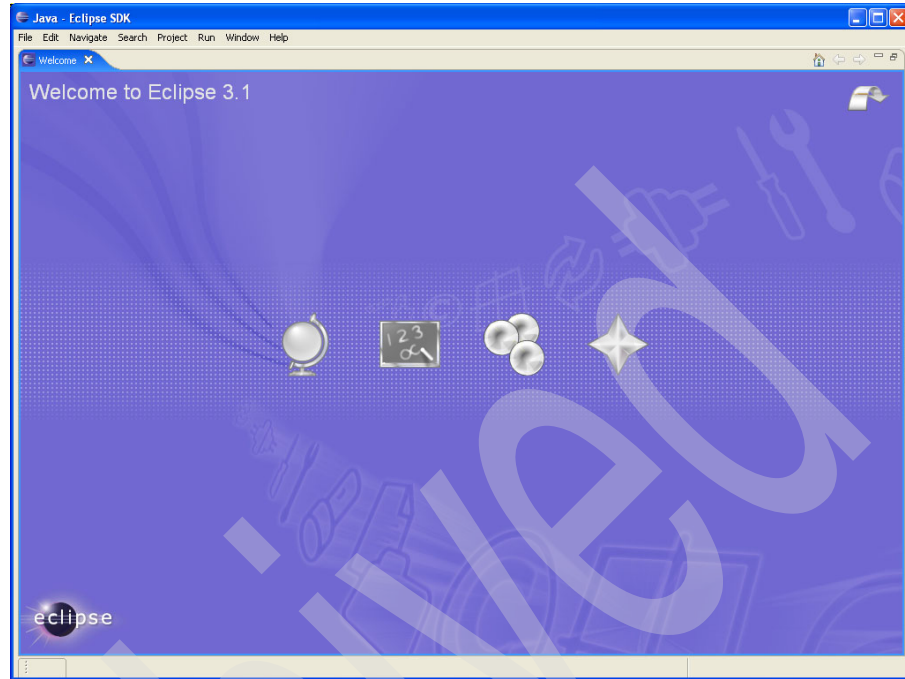


Figure 5-4 The Eclipse welcome screen

2. At the top right of the screen is the Go to the Workbench button (Figure 5-5). Click it to hide the welcome screen and go to the main workbench view.



Figure 5-5 The Go to the workbench icon

This automated procedure requires Web access. If you are behind a proxy server perform the following steps to set up this access:

- a. Select the menu option **Window** → **Preferences**.
- b. Select the **Install/Update** option on the left of the dialog box.
- c. Select the **Enable HTTP proxy connection** check box.
- d. Fill in your site-specific proxy address and port number (Figure 5-6 on page 118).

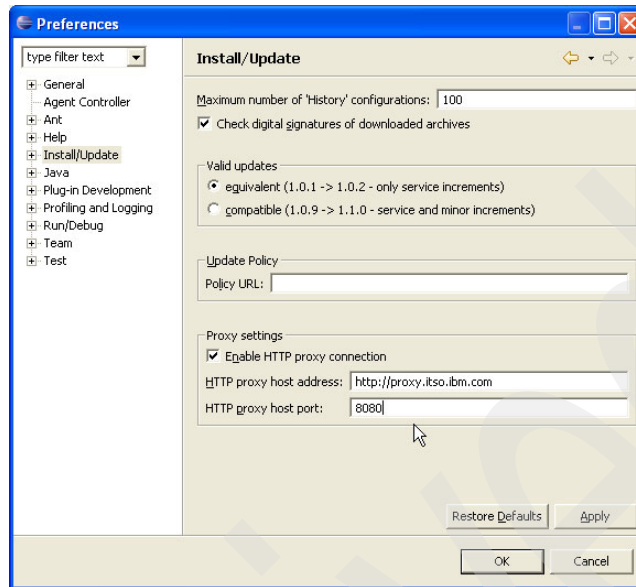


Figure 5-6 Eclipse Install/Update proxy settings

3. Select **Help** → **Software Updates** → **Find and Install** (Figure 5-7).

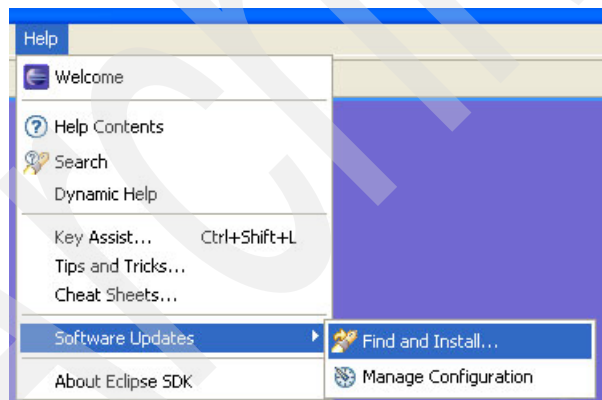


Figure 5-7 Select Find and Install

4. Select **Search for new features to install** and click **Next** (Figure 5-8).

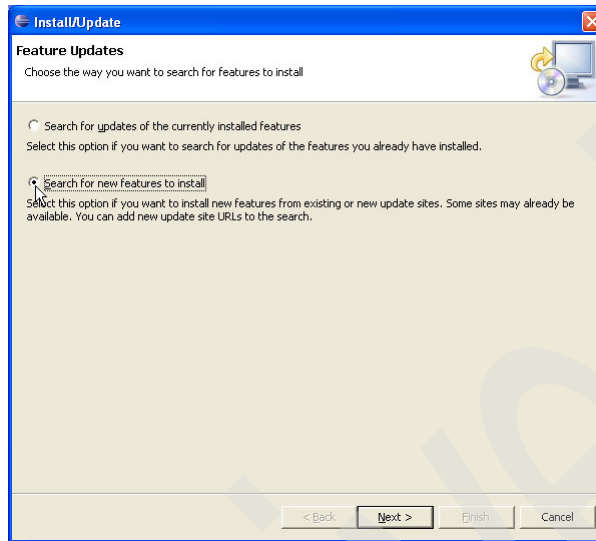


Figure 5-8 Selecting Search for new features to install

5. On the next dialog select the Eclipse.org update site and click **New Remote Site** (Figure 5-9).

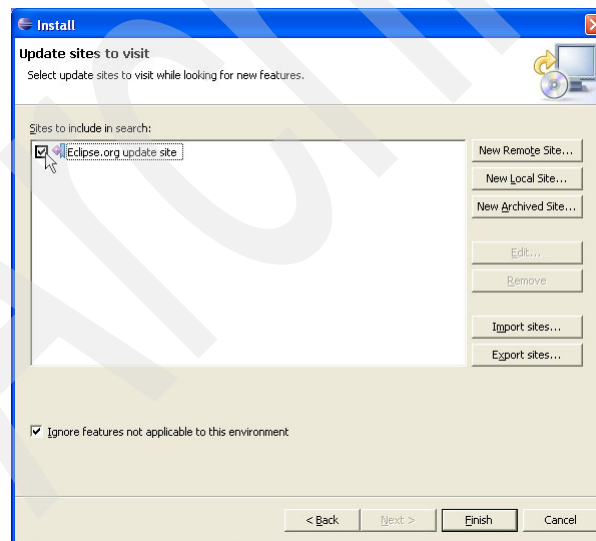


Figure 5-9 Update sites to visit

6. The New Update Site dialog is displayed. Set the Name field to TPTP Update Site and the URL field to:

`http://eclipse.org/tptp/updates/site.xml`

Click **OK**.

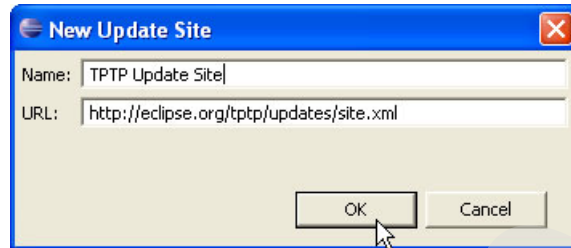


Figure 5-10 The New Update Site dialog

7. You now return to the previous dialog. Repeat the steps for the EMF update site, which is at:
`http://download.eclipse.org/tools/emf/updates/site.xml`
8. At the main dialog select the new update sites you have just added and click **Finish** (Figure 5-11).

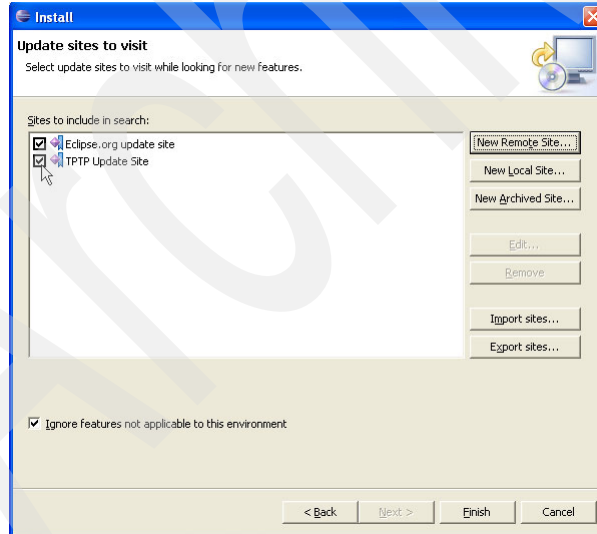


Figure 5-11 Selecting the update sites to search

9. The update manager will then search these sites for new features to install. It may display one or more mirror selection dialogs from which you can either

leave the main update mirror selected or select a local mirror to improve download speed (Figure 5-12).

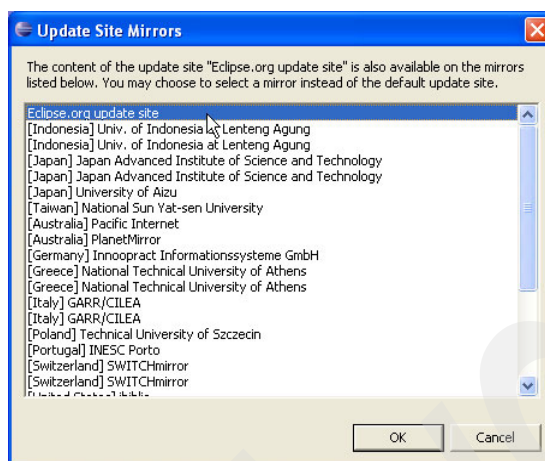


Figure 5-12 Update site mirror selection

10. When the search has completed you are presented with a list of potential new features to install. Expand the Eclipse.org Update site tree and select the **EMF SDK**, then expand the TPTP Update Site tree and select **TPTP Features**. Click **Next**. (Figure 5-13).

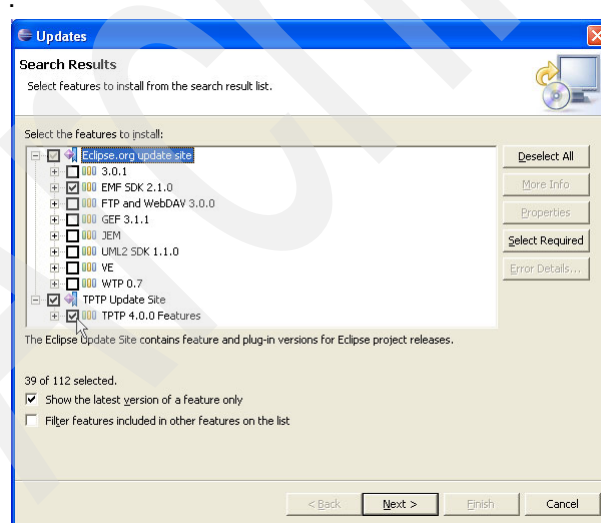


Figure 5-13 Selecting EMF and TPTP

11. After you read and accept the Eclipse license agreement the update manager lists the features that will be installed. Click **Finish** (Figure 5-14).

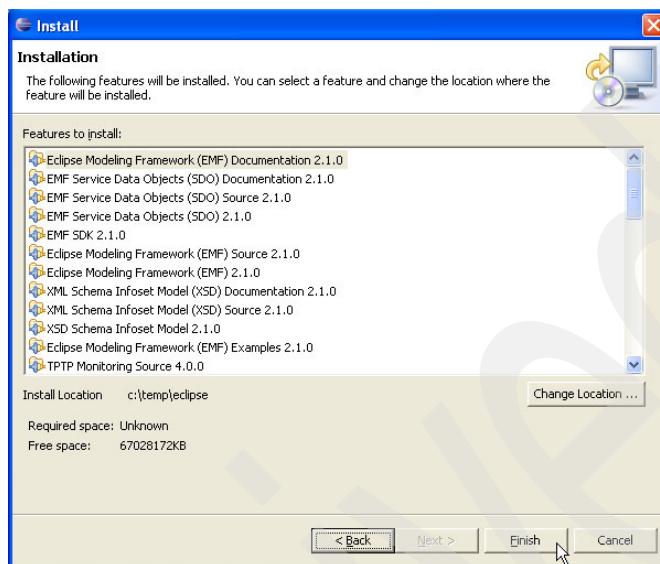


Figure 5-14 Features that will be installed

12. During installation, Eclipse may display a dialog explaining that a feature is not digitally signed. Because these features were downloaded directly from the Eclipse update site, click **Install All**.

The Update manager installs and configures EMF, XSD, and TPTP. At the end of this process it will ask if you wish to restart the Workbench. Click **Yes**.

Verifying the TPTP installation

Once the Eclipse workbench has restarted, you can verify that TPTP has been installed correctly by using the following procedure.

You should see the profiling icon  in the tool bar at the top of the screen.

If you cannot see this icon, select the menu option **Help** → **Software Updates** → **Manage Configuration**.

This will display all of the installed Eclipse features in a tree list. Expand the tree and look for the EMF, XSD, and TPTP features.

Click the **Show Disabled Features**  icon in the toolbar.

If any of the required features are installed and marked as disabled, right-click the features to enable them.

Note: Under Linux you may find that all of the new features are marked as disabled.

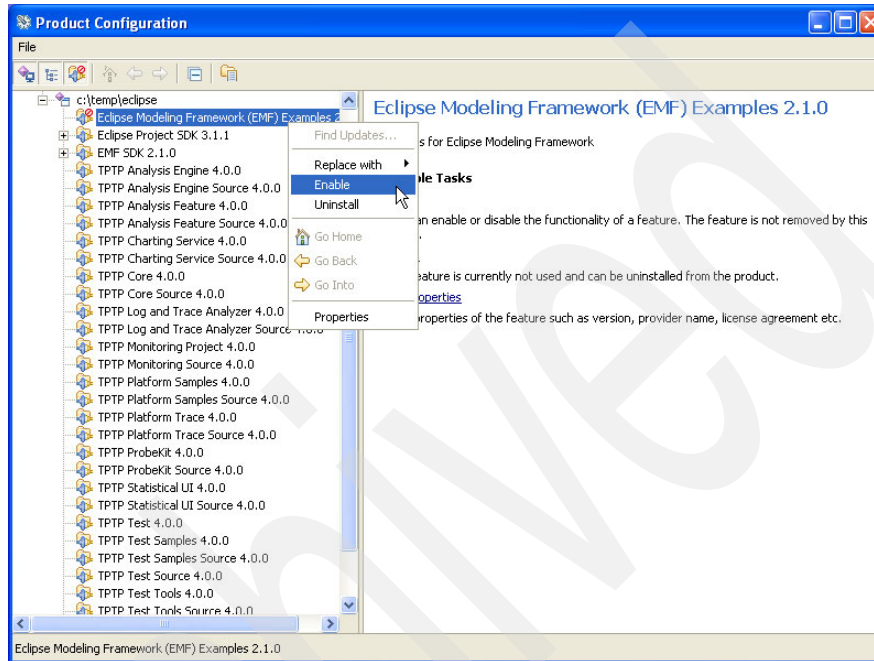


Figure 5-15 Enabling disabled features

The client side of Eclipse TPTP is now installed. The next task is to install the Remote Agent Controller on z/OS.

Installing the Remote Agent Controller on z/OS

The Remote Agent Controller on z/OS enables communication to and from the client workstation, launches or connects to already running JVMs being profiled, and collects the profiling data. This feature must be installed to enable you to profile a Java application running on z/OS. The Agent Controller is a daemon process submitted from a USS shell (or BPXBATCH). It is able to service multiple simultaneous requests from a number of clients. The architectural overview of the Remote Agent Controller is in Figure 5-16 on page 124.

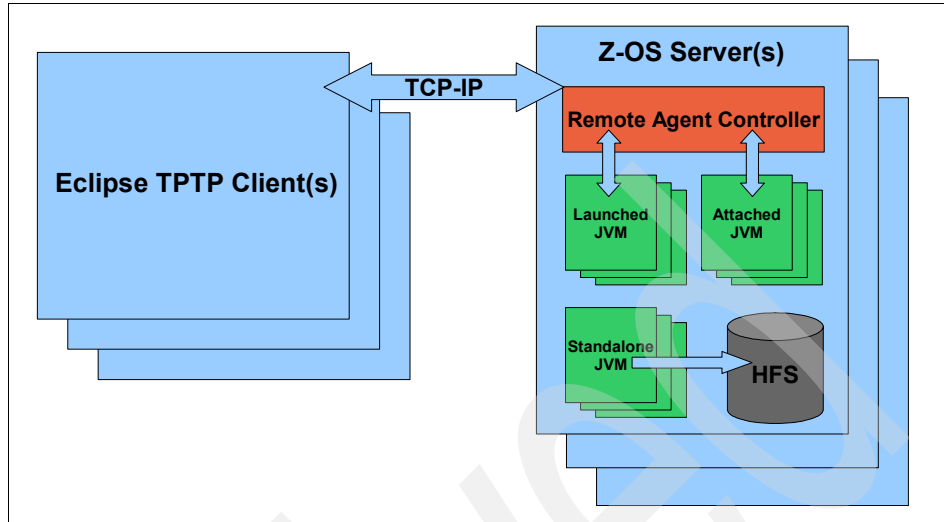


Figure 5-16 Agent Controller architectural overview

To install the Remote Agent Controller, return to the Eclipse TPTP page at:

<http://www.eclipse.org/tptp>

Scroll down and select the downloads link; on the next page scroll down to the Agent Controller section and download the Agent Controller for z/OS-ZSeries.

A note posted to the z/OS Agent Controller indicates a dependency for the XML C++ Parser for z/OS. If you do not already have this installed click the link defined in the dependency note and follow the download and installation instructions.

Note: At the time of this writing the TPTP Agent Controller for z/OS version 4.1.0.1 had a dependency for version 1.4 of the XML C++ parser for z/OS. This can be found at the following Web address:

http://www-03.ibm.com/servers/eserver/zseries/software/xml/download/cparser_download.html

The downloaded Agent Controller may be in zip file format. If it is, and you do not have an unzip utility on z/OS, then you will need to convert it to a tar or pax file before uploading it to z/OS using one of the popular PC zip utilities.

When the tar or pax file has been uploaded to z/OS, create a directory called AgentController and extract the file, for example, as shown in Example 5-19 on page 125.

Example 5-19 Extracting the Agent Controller tar file

```
cd /u/user
mkdir AgentController
cd AgentController
tar -xvf /u/user/uploads/AgentController.tar .
```

Check and correct, if necessary, the permissions of the files within this directory. Specifically, you need to make sure that the files in the `bin` and the `lib` directory have execute permissions. You can use the commands in Example 5-20 to change the file permissions.

Example 5-20 Changing Agent Controller file permissions

```
cd /u/user/AgentController/bin
chmod -R 775 *
cd /u/user/AgentController/lib
chmod -R 775 *
cd /u/user/AgentController/config
chmod -R 664 *
```

Perform the base configuration of the Agent Controller by changing to the `bin` directory and executing the `SetConfig.sh` script (Example 5-21).

Example 5-21 Executing the SetConfig.sh script

```
cd /u/user/AgentController/bin
./SetConfig.sh
```

This script will pose several questions, including the location of various other components (Example 5-22).

Example 5-22 Configuring the Agent Controller

```
Specify the fully qualified path of "java" (e.g. /usr/java1.4/jre/bin/java):
Default>"/Z16RD1/usr/lpp/java/J1.4/bin/java" (Press <ENTER> to accept the
default value)
New value>
```

If The Default is OK press enter

```
Network access mode (ALL=allow any host, LOCAL=allow only this host,
CUSTOM=list of hosts):
Default>"LOCAL" (Press <ENTER> to accept the default value)
New value>
```

Enter ALL as we wish to allow all clients to connect

Please enter the JBoss Application Server Home:
Default>"" (Press <ENTER> to accept the default value)
New value>

We do not have JBoss installed, press enter

Please enter the JOnAS Application Server Home:
Default>"" (Press <ENTER> to accept the default value)
New value>

We do not have JOnAS installed, press enter

Note: The setup script asks for the location of any installed JBoss or JOnAS instances so it can hook into their performance monitors if they are available.

5.6.2 Profiling an application on z/OS

There are several main ways to profile an application on z/OS using TPTP. These are:

- ▶ Remote profiling of a launched JVM
- ▶ Remote profiling of an attached JVM
- ▶ Stand-alone profiling of a JVM, from the start
- ▶ Stand-alone profiling of a JVM with programmatic control

Remote profiling

This is the default mode of operation in which an Eclipse TPTP workbench connects over TCP/IP to a Remote Agent Controller (RAC). The RAC can be used either to launch a new JVM or to attach to a JVM that has already been started, with the correct command line parameters. As profile events are generated they are sent over Inter Process Communication (IPC) shared memory from the profiled JVM to the RAC, which buffers them and sends them to the profiling client. In addition to this, control requests from the client are sent to the RAC and the profiled JVM, for example to start and pause event profiling. The profile data is collected at the client, where it can be analyzed or saved for later analysis.

Standalone profiling

Remote profiling may not be ideal or even possible in some cases or environments. In those cases, TPTP can act in a stand-alone mode in which the JVM is started with certain command line parameters and profile data is collected to an XML file on the local file system for later analysis. One problem with stand-alone profiling is the lack of control over which data is collected and when

to start and stop profiling. The Java method events collected can be controlled via filtering and a filter file can be defined on the command line. As for when to profile, in its default mode, standalone profiling will collect from the very start of the JVM's execution until the application finishes or is stopped. This can produce a very large amount of trace data, which can be difficult to analyze later on. A more sophisticated method of starting and stopping event collection is via programmatic control, in which the application is adjusted and calls to the TPTP API are made to start and stop data collection, as illustrated in Example 5-23.

Example 5-23 An example of standalone TPTP programmatic control

```
import org.eclipse.hyades.collection.profiler.Profiler

public class myClass {

    .....

    public void myMethodOfInterest() {

        Profiler.getProfiler().startProfiling(true);

        ..... application code ....

        Profiler.getProfiler().stopProfiling();

    }

    .....

}
```

Filtering

For reasons of portability and openness the profiling data is generated and stored in XML. An application may call millions of methods and create hundreds of thousands of objects, so this data can become very large. This not only affects disk storage but also the performance of the client workstation during analysis. A way of managing the size of this data is by *filtering*, which allows you to define which packages, classes, and method calls you wish to include or exclude from data collection.

The performance of TPTP with large data sets has improved greatly recently, but some form of filtering may still be required to manage data volumes or be advantageous in helping focus on certain areas of the application.

A very important point to realize about modern Java applications is that the majority of code that is running is normally not written by the application

developers but rather is “included” by the use of 3rd party libraries. In some cases over 90% of the Java executing is from these ISV and Open Source libraries or the JDK itself. Filtering out all but the application code can lead to an artificial view of the execution and may prevent deeper analysis of an issue. For example, an application method that performs database updates may be seen to be taking a long time to execute, but with no further information due to filtering the real reason for this may not be visible. Including all method calls would highlight the cost of an Open Source persistence layer, for example, excessive calls to logging, or the JDBC driver itself.

Whenever possible, limited or no filtering should be used at some point during profiling to expose the probable contribution to the performance of an application by included libraries or the inefficient use of these libraries by the application.

Accuracy and overhead

TPTP utilizes the Java Virtual Machine Profiling Interface (JVMPI). The JVM calls the native TPTP code when a significant event occurs, such as method entry and exit, object creation and garbage collection. These calls out of the JVM into TPTP naturally incur an overhead, which TPTP attempts to calculate and remove from the reported elapsed and CPU time. So, in theory, although the real execution time during profiling may be increased greatly, the elapsed and CPU time reported by TPTP should be similar to the real figures achieved without TPTP; in practice, however, this is often not the case. This is due to many factors. For example, any error or variability in the calculated overhead is multiplied for each event, so in a section of code that may call a million methods this error is multiplied many times. In addition, the fact that the JVMPI interface is enabled can prevent the JVM from making some of the more exotic optimizations it is capable of. Finally, processor caches can become less effective, further affecting the performance of the application. Generally, however, profiling can be trusted to give a good impression of the relative costs of certain components of the application, as well as a higher level insight into architecture, efficiency, flow, and the hidden costs of vendor, third party, and JDK code.

Starting the Remote Agent Controller

To be able to either launch or attach to a JVM process for profiling on z/OS from your client, you need to start the Remote Agent Controller that you installed previously. This is done by executing the RASStart.sh shell script from within the ./AgentController/bin directory, but before you can do this you need to change an environment variable. The RAC depends on the XML C++ parser for z/OS, which you have also already installed. You need to change the LIBPATH environment variable to point to the shared libraries from this product, as shown in Example 5-24 on page 129.

Example 5-24 Adding XML for C to LIBPATH

```
export LIBPATH=$LIBPATH:/usr/lpp/xml4c-4_0/lib
```

This setting can be made permanent by adding it to your user profile or the system profile, adding it to the RASstart.sh shell script, or by creating a new shell script to wrap this change and then call RASstart.sh.

Now you can start the Remote Agent Controller by executing RASstart.sh, as shown in Example 5-25.

Example 5-25 Starting the Remote Agent Controller

```
cd /u/user/AgentController/bin
./RASstart.sh
```

This should produce output similar to that shown in Example 5-26.

Example 5-26 Output from a successful Agent Controller start

```
Starting Agent Controller
RAServer started successfully
```

The RAC is now running and should be listening for requests from TPTP clients on the default ports 10002 and 10005. This can be verified by executing the netstat command, as shown in Example 5-27.

Example 5-27 Checking that the Agent Controller is listening for connections

```
netstat
.....
OMVSKERN  0000A525  0.0.0.0:10002      0.0.0.0..0      Listen
OMVSKERN  0000A525  0.0.0.0:10005      0.0.0.0..0      Listen
.....
```

If the default ports need to be changed in your environment, this can be achieved by manually adjusting the configuration and restarting the RAC. The file that must be changed is ./AgentController/config/serviceconfig.xml.

Example 5-28 Changing the default agent controller ports

```
~:> cd /u/user/AgentController/config
~:> vi serviceconfig.xml
```

Example 5-29 The serviceconfig.xml file

```
<?xml version="1.0" encoding="ibm-1047-s390"?>
<AgentControllerConfiguration activeConfiguration="default" filePort="10005">
```

```
isD
ataMultiplexed="false" jvm="/usr/lpp/java/J1.4/bin/classic/libjvm.so"
loggingDet
ail="LOW" loggingLevel="INFORMATIONAL" port="10002" processPolling="true"
securedPort="10003" version="4.1.0">
```

Shared memory configuration

The Remote Agent Controller uses shared memory to communicate between itself and any profiled JVMs, and this may require additional configuration on z/OS to work correctly.

The size of the shared memory segments is defined in the file `pluginconfig.xml`, which is located under the RAC directory in `./plugins/org.eclipse.tptp.platform.collection.framework_4.x.x/config/`.

Example 5-30 Shared memory configuration in pluginconfig.xml.

```
....
<Agent client="DYNAMIC" configuration="default" dataChannelSize="32M"
name="Java Profiling Agent" type="Profiler"/>
....
```

On z/OS, the BPXPRMxx parameters IPCSHMSPAGES, IPCSHMMPAGES and MAXSHAREPAGES control the amount of shared memory available. They need to be set high enough to accommodate the size defined in `pluginconfig.xml`.

In addition, the two USS commands `ipcs` and `ipcrm` can be used to query and manually clean up shared memory in the case of a failure.

Launching and profiling an application

Now that the RAC is started, you can return to the client workstation and start Eclipse. In this section we describe how you can launch and profile a simple Java application that has already been deployed to z/OS by using ftp to transfer the jar file to an HFS directory.

In Eclipse, select the profiling icon in the toolbar by clicking on the downward facing triangle, and then selecting the **Profile** sub-option (Figure 5-17).

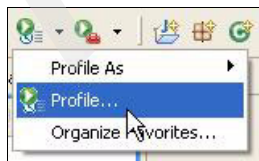


Figure 5-17 Opening the Profile dialog in Eclipse

This will launch the Profile Configuration dialog. Select **External Java Application** from the navigation pane and click **New** (Figure 5-18).

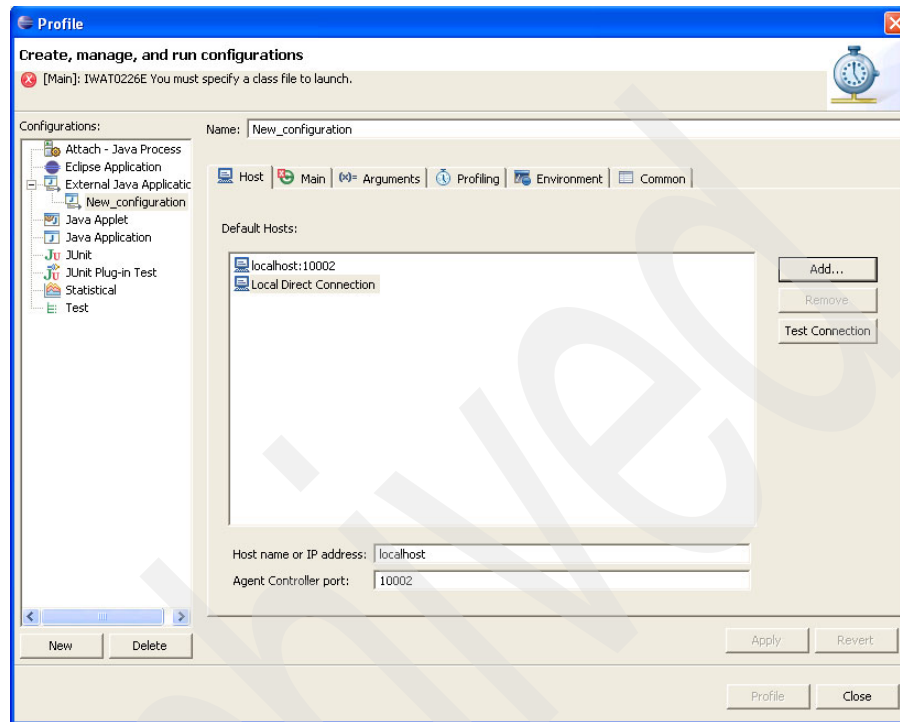


Figure 5-18 Creating a new profiling configuration

Click **Add** and enter the hostname or IP address of the z/OS system running the Remote Agent Controller. Change the port number if you are not using the default (Figure 5-19).

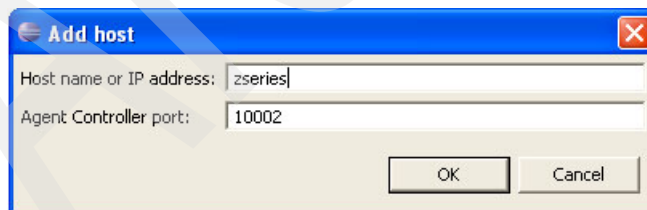


Figure 5-19 Creating a new connection to a host

Select this new entry and click **Test Connection**.

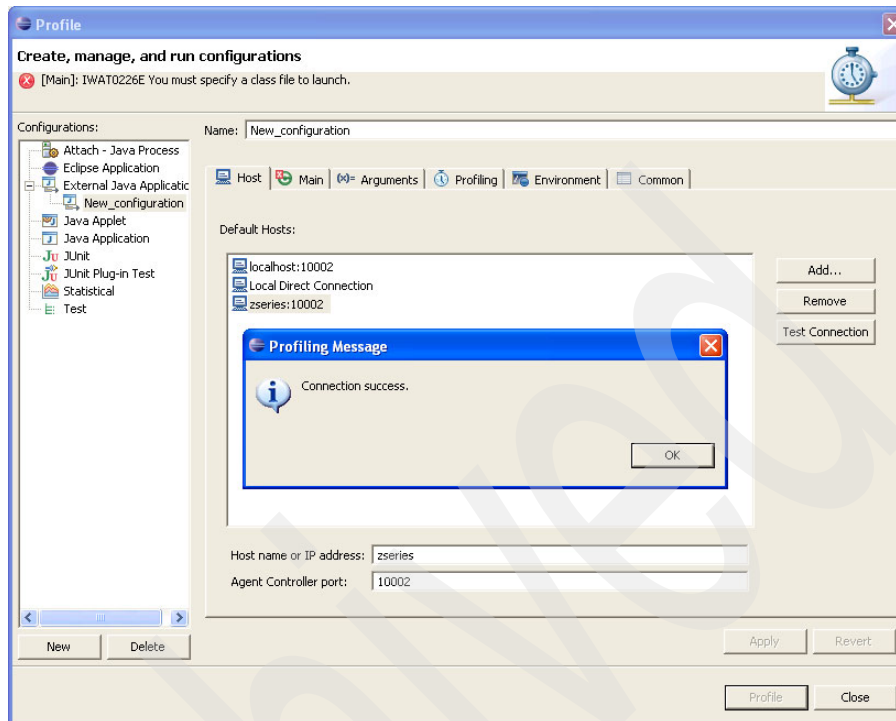


Figure 5-20 A successful connection test

If the connection is successful a notification will be returned. If the connection fails, which may take a while to report, check that the RAC is running on the server and check the address and port settings.

The Java program we are profiling in this discussion is the classic Hello World example.

Example 5-31 The Hello.java example to be profiled

```
public class Hello {

    public static void main(String[] args) {
        System.out.println("Hello Profiling World!");
    }

}
```

This Java class needs to be compiled and uploaded to a directory on z/OS, in our case /u/user/hello/.

After the connection has been tested, click the next tab in the dialog, Main. This is where you define the Java program that you wish to launch and profile.

Define the Main class as Hello and the classpath as /u/user/hello/ (Figure 5-21).

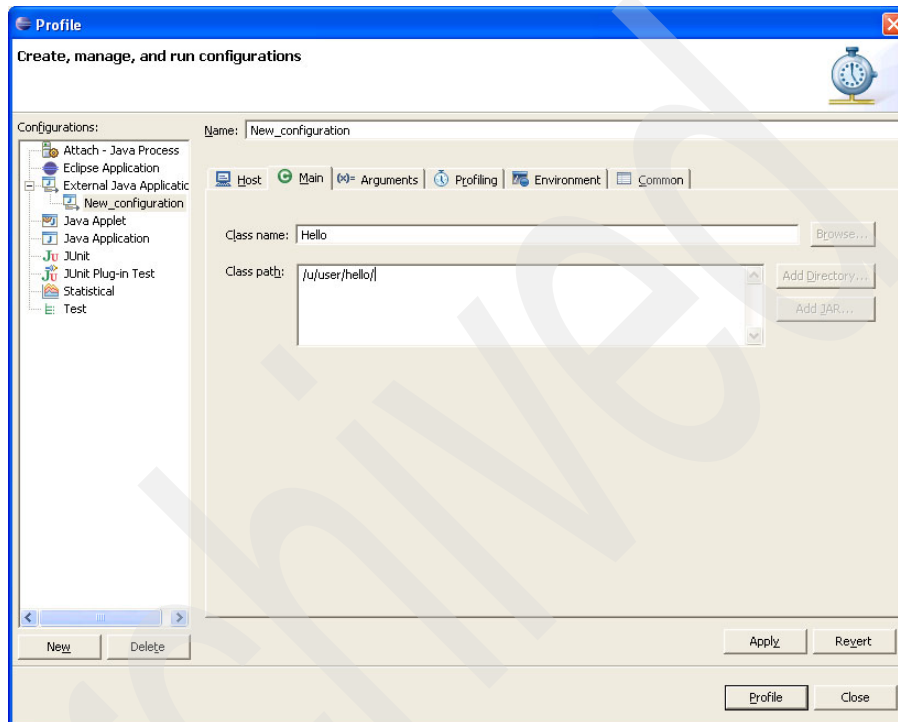


Figure 5-21 The Main tab in the profiling dialog

The next tab, Arguments, allows you to define any program or JVM arguments as well as set the working directory. You can ignore these for this simple test.

The next tab, Profiling, is the most important one and itself has three sub tabs: Overview, Limits, and Destination (Figure 5-22).

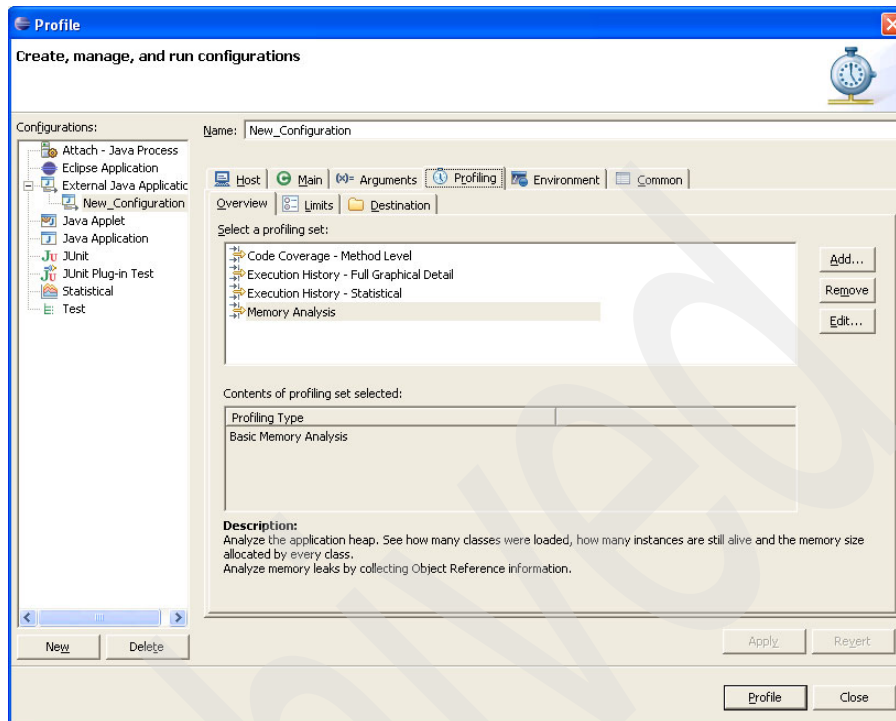


Figure 5-22 The Profiling/Overview tab and profiling sets

In the Overview sub tab you are presented with several pre-defined *profiling sets*. These are configurations of which profiling data to record.

TPTP has three broad categories of profiling information that it can collect for an application. These are discussed in the following sections.

Code coverage

This information shows you the number of functions executed in the application as a percentage of the total number of functions in the packages and classes. For example, if you were to have a package with 100 classes, each with 10 methods, then understanding the percentage of methods exercised in a profile would be important in understanding whether further analysis of the under utilized functions was needed. Confidence in the performance and stability of the code can be increased by guaranteeing that as much as possible of the application had been tested. Alternatively, it may lead to refactoring of the code.

Memory analysis

As the title suggests, TPTP can analyze the memory utilization of an application to assist with leak analysis and to reduce long-term and temporary memory

usage. Garbage collection behavior can be reviewed and object reference patterns browsed.

Time analysis

This is often the most heavily used TPTP profiling data because it gives a clear and direct understanding of the performance and flow of an application down to the method level. The data collected can be statistical in nature or graphical. Additionally, CPU as well as elapsed times can be measured.

You can define your own profile set to collect graphical time data. To do this, click **Add** on the Overview sub tab. You are presented with a dialog; enter a name and optionally a description for this new profiling set, for example Hello Analysis. Click **Next**.

Deselect the **Memory Analysis** option in the tree in the left pane (if it is selected), select the **Time Analysis** option, and expand this tree by clicking the plus sign next to it.

Now that Time Analysis has been expanded you should see beneath it **Execution Time Analysis**. Make sure that it is also selected and click it to display the dialog shown in Figure 5-23 on page 136.

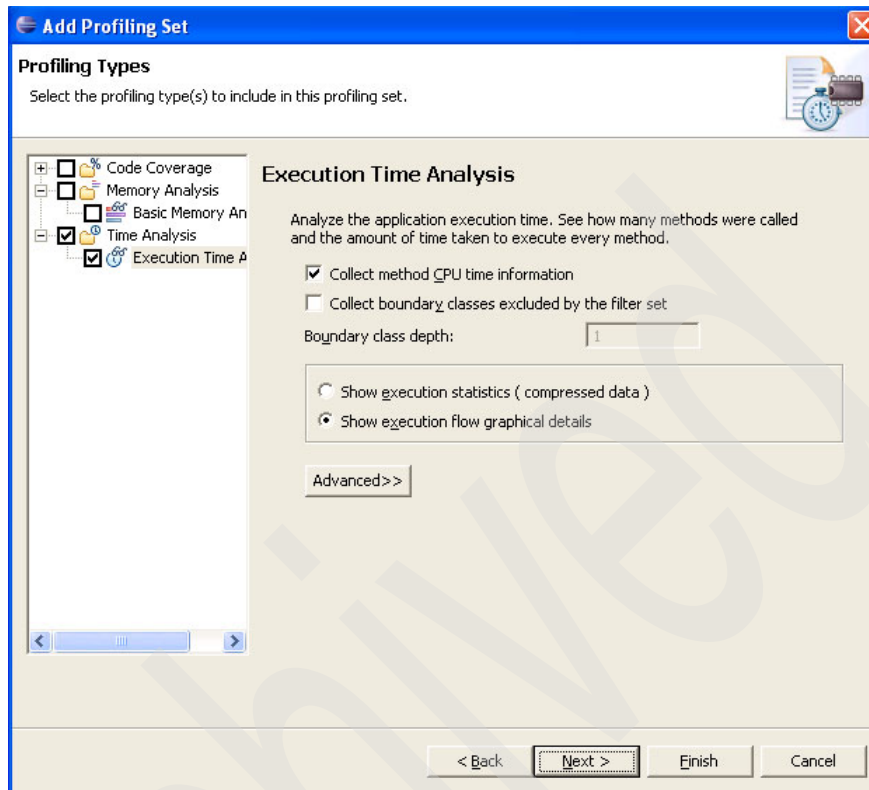


Figure 5-23 Creating a new profiling set

On the Execution Time Analysis page, select **Collect method CPU time information** and **Show execution flow graphical details**. Click **Next**.

The next dialog allows you to select or define a filter set. There should be three pre-defined filter sets. In this test you want to collect all method data, so you create a new filter set. Click **Add**, enter the name Everything for this new filter set, and click **OK**. Your new filter set is added to the list and automatically selected. At the moment it contains default filters that would, if left in, exclude all of the JDK function from the trace.

Select the first of these EXCLUDE filters in the list and then click **Remove**. Continue to click **Remove** until the list is empty. Your new filter set will now make sure that every single method call executed in the profiled JVM will be recorded. (Figure 5-24 on page 137).

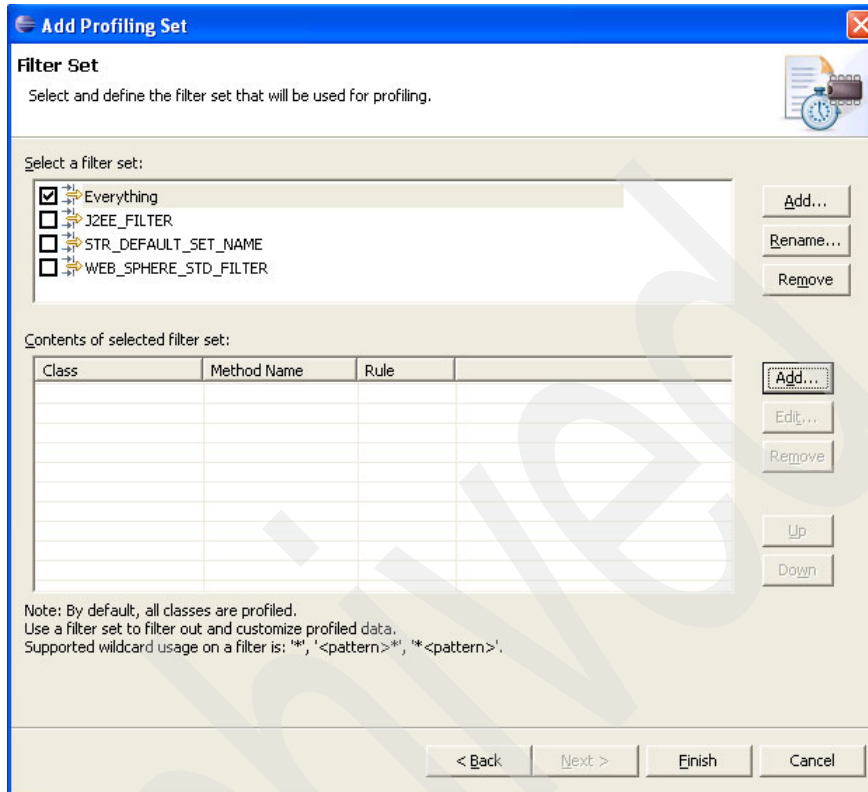


Figure 5-24 Removing the default EXCLUDES from the new filter set

Click **Finish** on this dialog; you return to the Overview sub tab. Click the next sub tab, Limits. This is where you define limits on the amount of profiling data to be collected or the length of time for the collection to run. This helps control the potentially large amount of data that the profiling workstation has to deal with. The example program is very small, so you can skip these definitions, but at the bottom of the dialog there is a check box that you should select – Automatically start monitoring when application is launched. Because the example application will execute very quickly, you would not have enough time after launching it to manually start data collection before it is completed (Figure 5-25 on page 138).

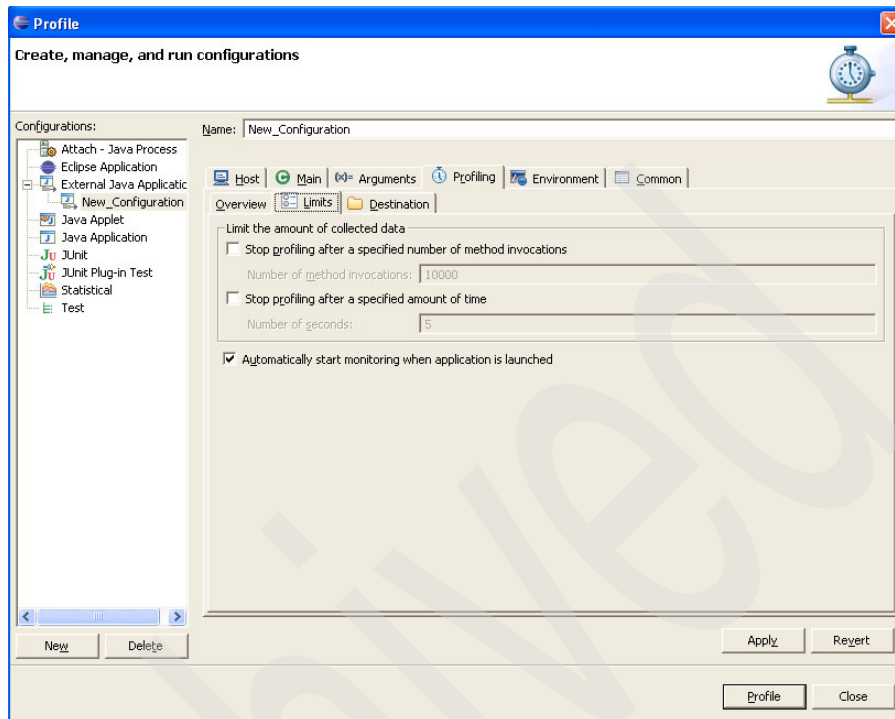


Figure 5-25 Selecting to automatically start monitoring on the Limits sub tab

The final sub tab, Destination allows you to define the project structure in the workbench where this profiling session and its data will be stored, and whether to save the data explicitly to an external file on disk. Skip this sub tab, leaving the defaults.

Going back to the main tab group, there are two remaining: Environment, where you can define OS environment variables, and Common, which contains the remaining settings. For this example, you can leave these at default values.

Before you click the **Profile** button to actually launch the Hello program remotely on z/OS, give this new configuration a name by entering it at the very top of the dialog. Hello seems appropriate; then click **Apply**.

Launching

You are now ready to launch and profile the application. Click **Profile** on the dialog. When asked if you wish to switch to the profiling and logging perspective, click **Yes**.

On the left of the Profiling Monitor view is a tree structure representing this profiling session (Figure 5-26).

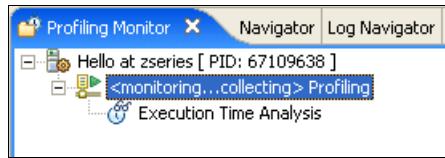


Figure 5-26 The launched profiling session in the Profiling Monitor

Double-click the **Execution Time Analysis** leaf. This opens the default view for this data type, which is Execution Statistics (Figure 5-27).

>Package	Base Time (sec...)	Average Base T...	Cumulative Tim...	Calls
(default package)	0.014660	0.014660	0.099618	1
[byte]	0.000000	0.000000	0.000000	0
[char]	0.000000	0.000000	0.000000	0
[int]	0.000000	0.000000	0.000000	0
[byte]	0.000000	0.000000	0.000000	0
[char]	0.000000	0.000000	0.000000	0
Hello	0.014660	0.014660	0.099618	1
main(java.lang.String[])	0.014660	0.014660	0.099618	1
int	0.000000	0.000000	0.000000	0
com.ibm.jvm	0.000222	0.000002	0.000222	91
com.ibm.jvm.io	0.000000	0.000000	0.000000	0
java.io	0.190962	0.000554	0.246654	345
java.lang	0.253351	0.000047	1.349107	5363
java.lang.ref	0.011767	0.000168	0.011777	70
java.lang.reflect	0.003881	0.000970	0.029479	4
java.net	0.111763	0.000202	1.277071	553
java.security	0.085691	0.000912	1.263231	94
java.util	0.108642	0.000172	0.149412	631
java.util.jar	0.155557	0.001275	0.358665	122
java.util.zip	0.083755	0.000153	0.125901	547
sun.io	0.009722	0.001620	0.009722	6
sun.misc	0.238531	0.000887	1.318176	269
sun.net.www	0.070201	0.003343	0.078948	21
sun.net.www.protocol.file	0.057755	0.008251	0.094083	7
sun.net.www.protocol.jar	0.006282	0.006282	0.006319	1
sun.nio.cs	0.018181	0.001515	0.048910	12
sun.reflect	0.027241	0.003027	0.033563	9
sun.security.action	0.000451	0.000038	0.044762	12

Figure 5-27 The default execution statistics view

In this view you can see execution time statistics grouped by package, class, and method. You can view the figures as absolute seconds values or as percentages of the whole.

Expand the (default package) tree, then expand the Hello class and select the **main** method. You can see the number of times it was called, its *base time*, which is the time it took itself, and its *cumulative time*, which is the total of its own time and the time of any methods it called.

Double-click the `main` method. The Invocation Details panel is displayed. In the case of the `main` method it was invoked by nothing, as it is a main method, and it invoked the `java.io.PrintStream.println` method.

Close the method invocation details; this returns you to the execution statistics view. Experiment with the Execution Statistics tool bar at the top right (Figure 5-28). It allows you to view by package, class, or method; to show the value as a percentage; and to open any attached source. There is also a simple reporting feature.



Figure 5-28 The Execution Statistics toolbar

Returning to the Execution Time Analysis leaf, right-click it and from the pop-up menu select **Open With**, then **Execution Flow**. A new view will be opened and you should see a colored graphical display like the one in Figure 5-29 on page 141.

This visualization of the execution flow of a Java application is an extremely powerful tool. It can take a while to adjust to viewing profiling data in this way, but once you become familiar with the concept it will probably become your view of choice.

This powerful visualization technology was invented in 1998 by Wim De Pauw of the IBM's TJ Watson Research Center in New York. It was the core component of IBM's Jinsight profiling tool and has been migrated to WebSphere Application Developer, Rational Application Developer, and now Eclipse TPTP. Visit <http://www.research.ibm.com/jinsight> for more information. Wim continues to explore advanced software visualization techniques for IBM, breaking new ground in this important field.

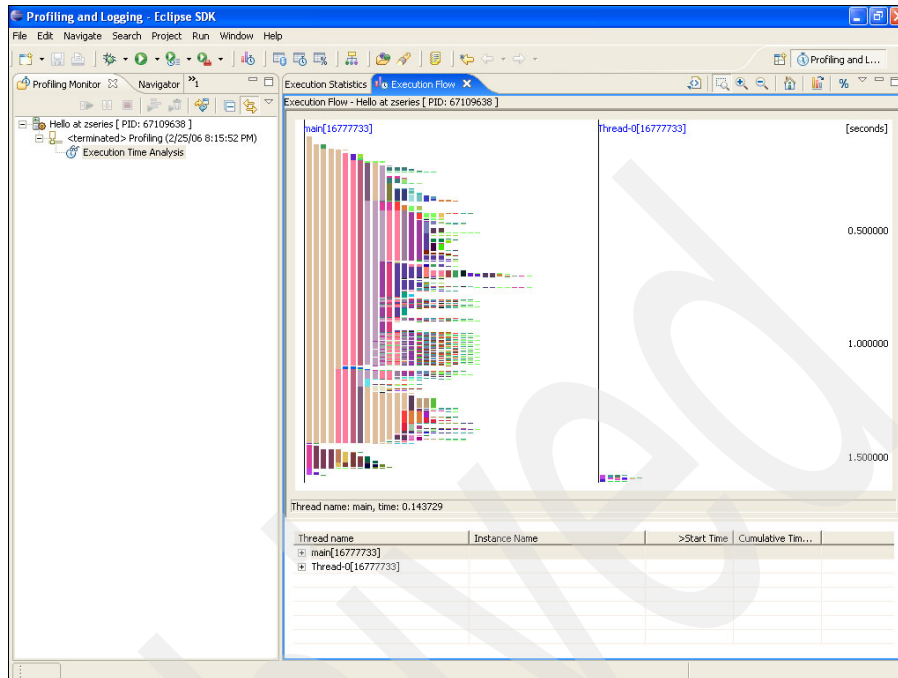


Figure 5-29 The Execution Flow view

Double-click the tab title to maximize this view. In the main graphical display you should see two thin vertical lines marked at the top as `main[nnnnn]` and `Thread-0[nnnnn]`. These represent the threads that were executing during the profiling period. In this example we are only interested in the main thread and we will hide the others by right-clicking the view, selecting **Threads** from the pop-up menu, and then deselecting **Thread-0[nnnnn]**.

Go to the tool bar for this view and click the **Home**  icon.



Figure 5-30 The Execution Flow toolbar

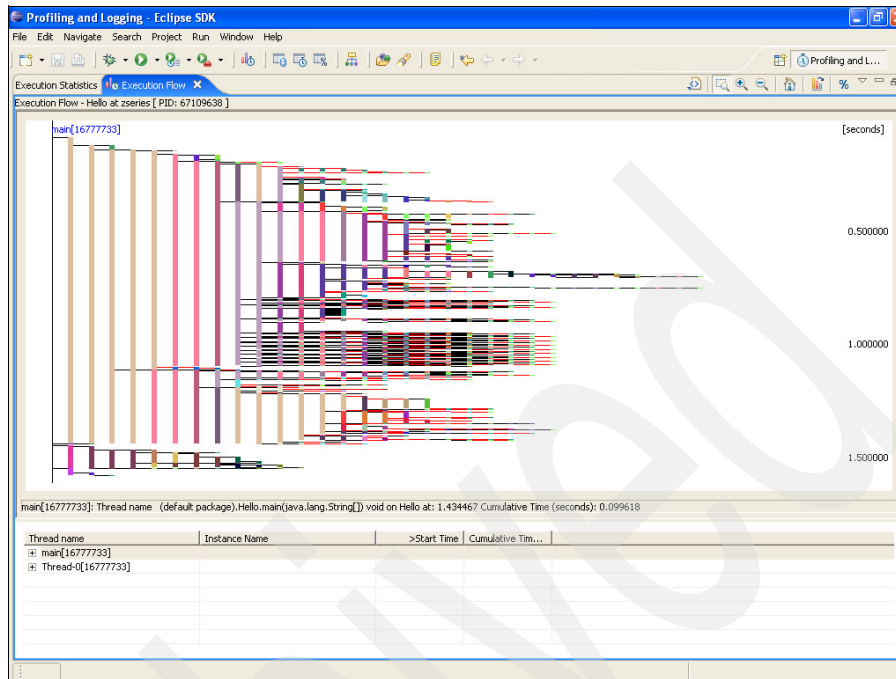


Figure 5-31 The Execution Flow view showing only the main thread

The Execution Statistics tab is still open but it is now behind the Execution Flow tab. Clicking on a tab's title brings it to the foreground. It is also possible to detach these tabs completely from the main Eclipse window. This can be done by right-clicking the tab title and selecting **Detached**. Alternatively, with part of the desktop visible, click and drag the Execution Flow tab out of the main Eclipse window. This will open a separate window just for this view, increasing the available area for the visualization display. Clicking and dragging the tab title back to the main Eclipse window and placing it beside the Execution Statistics title returns it to its original position.

Execution flow basics

The visualization technology used in the execution flow is designed to exploit the extensive pattern recognition capabilities of the human brain to convey information. "A picture is worth a thousand words" goes the old saying, and in this case it can be worth pages of raw numbers.

One of its greatest benefits is that it can be used to examine very low-level implementation as well as much higher level flow and architecture.

Figure 5-32 is an example of the basic structure of the execution flow.

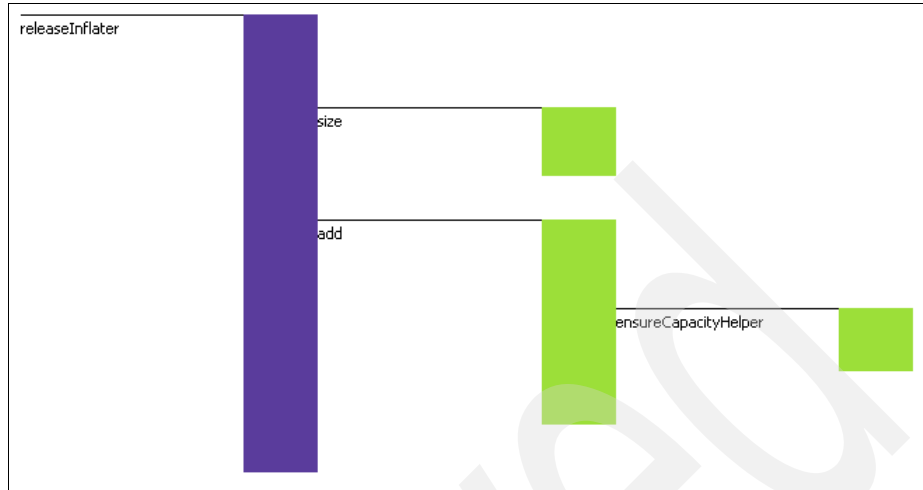


Figure 5-32 A simple example of the structure of the Execution Flow

In this view, looking from left to right, there is a thin horizontal black line with the name of a method, **releaseInflater**. Beneath it, this form indicates a method call. In this case it would seem to originate from nothing; in fact, the parent has been removed to simplify this explanation.

This thin horizontal line runs from left to right and terminates in a thick vertical blue bar of a certain height. This bar indicates the length of execution of the **releaseInflater** method, that is, the time index runs vertically from top to bottom. There is no information imparted in its thickness, which is constant and related to the level of magnification.

Branching from the vertical blue **releaseInflater** method are two other method calls, again indicated by the thin horizontal line with the method name beneath it. First, we have the **size** method, represented this time by a green vertical bar that appears to execute for less time, that is, the bar is shorter. The **size** method bar then terminates and the next event, reading down, is the call to the **add** method, which means that the **size** method returned to **releaseInflater** for it to continue its processing. Method call returns are not explicitly indicated by a returning thin horizontal black line.

So, **size** returned to **releaseInflater**, which then went on to call the **add** method as previously described. The **add** method itself calls a method, **ensureCapacityHelper**, which executes and returns to **add**. Then **add** returns to **releaseInflater**, which finishes processing and returns to the parent we cannot see.

So in effect the execution flow is a display of the call stack over time.

The colors used for the method bars also contain important information. In its default setting TPTP will choose a set of colors for the display using the same color for all of the methods of the same class.

In this example the **releaseInflater** method is part of the `java.util.zip.ZipFile` class and has its own color. The **size**, **add** and **ensureCapacityHelper** methods are all part of the `java.util.Vector` class and therefore share the same color.

Unfortunately, at the moment TPTP will change these colors every time a profile is opened; Jinsight used to, and still does generate colors based on the hash of the package and class name. Therefore, they would be consistent across analysis sessions on the same trace file and even across different trace files from different platforms.

Another important visual form in the execution flow display is used to indicate Object construction.





Figure 5-33 A sample Execution Flow including Object construction


In Figure 5-33 you can see the **toString** method of the `StringBuffer` class. Branching from it you can see a thin horizontal red line with the name **String** underneath it, also in red, which indicates a call to a constructor method of **String** and therefore the creation of `String` object. This `String` itself constructs a new `java.lang.Object` and then goes on to call the **setShared**, **getValue**, and **length** methods on the `StringBuffer` object that was passed to it as a parameter.


Navigating the execution flow

The navigation of the execution flow is fairly straightforward, but it takes some practice. Basically it is a canvas that can be zoomed in and out of and moved left and right, allowing you to examine any part of the flow in any level of detail.


The Select Zoom  tool enables a mode where the mouse can be used to click and drag a bounding box around the area of interest in the display. Releasing the mouse causes the view area to zoom in on the contents of the drawn box.

The Zoom In  tool zooms the entire display evenly in with the center point of the zoom at the point in the view area where the mouse is clicked.

The Zoom Out  tool zooms the entire display evenly out with apparently no particular center point when clicked on the view area.


The Home button  when clicked returns to a zoom level that fits the entire trace optimally in the available display area.

Highlighting is an important feature of the display and assists analysis. By clicking on a method in the display, it and its called methods are highlighted in yellow. You can then right-click the highlighted method, and from the pop-up or context menu, select **Show Method Invocation**. This opens a new Execution Flow view tab for just this method invocation and any methods it calls.

Tip: You may wish to choose a stronger shade of yellow for highlighting to make very small selections more visible. This can be done by clicking the Graph Colors  tool, and changing the Selection color.

You might find that some or all methods are not clickable. This would appear to be a bug in the current level of TPTP. There is a way to obtain the same results by using the alternative method of highlighting described later in this section.

The Execution Statistics tab should still be open; if it is not, double-click the Execution Flow tab title to return it to its former size and right-click the Execution Time Analysis leaf of the profiling session tree. From the context menu select **Open With → Execution Statistics**.

In the Execution Statistics view use the package button  in the tool bar to order by package. At the top you will see (default package). Expand it, then expand the Hello class and select the main method.

Return to the Execution Flow view by double-clicking its tab title to maximize the view. Click the **Home** button; you will see at the bottom of the trace that our **main** method has been highlighted in yellow. (Figure 5-34 on page 146).

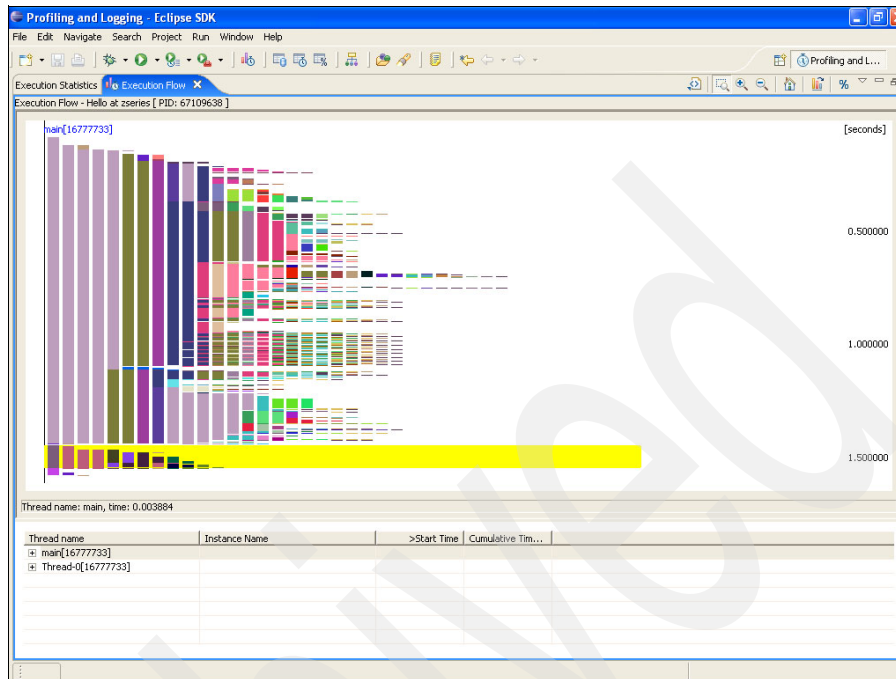


Figure 5-34 The Execution Flow with the main method highlighted in yellow

Now using the Select Zoom tool, zoom in on the highlighted area and right-click the highlighted main method. From the displayed context menu select **Show Method Invocation**. This opens up a new view tab in which only the main method and all of its callees are displayed as an execution flow (Figure 5-35 on page 147).

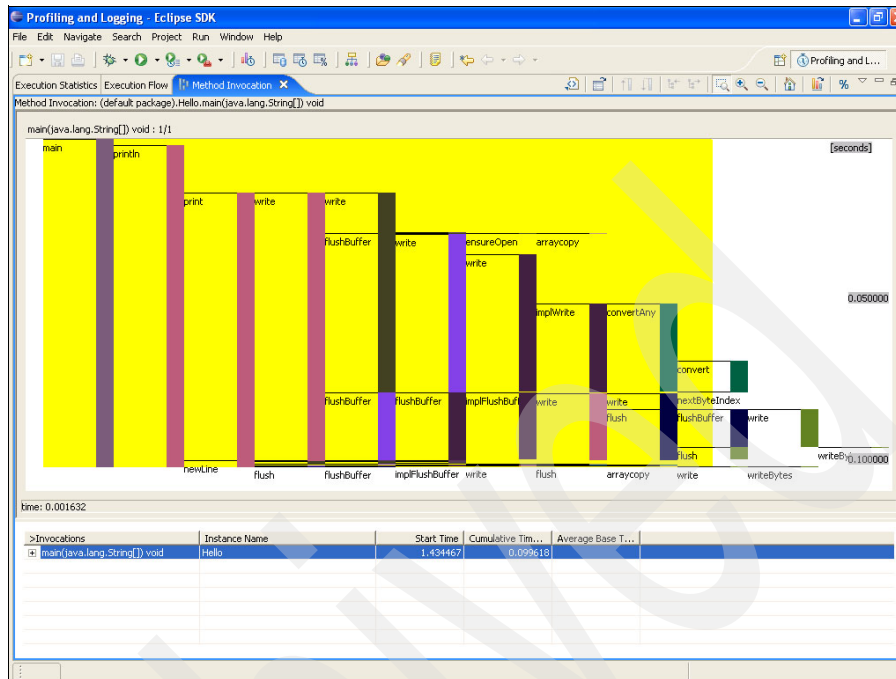


Figure 5-35 The isolated main method Execution Flow

From this display, again right-click the main method and from the context menu select **Method Invocation Details**. This brings up the details view for this method invocation in a new tab.

If you now move between the tabs, selecting different methods, you will see that they are linked, and a change in selection of one is reflected in the other.

The views can be rearranged or detached, and on a single workstation a very productive analysis workbench can be constructed (Figure 5-36 on page 148).



```

        while(true) {
            String message = new String("Hello Profiling World! " + count)
            System.out.println(message);
            count = new Integer(count.intValue() + 1);
            try {
                Thread.sleep(500);
            } catch (Exception e) {
            }
        }
    }
}

```

To launch this class with profiling enabled, the command line must contain the directive **-XrunpiAgent**, which causes the JVM to prepend *lib* and append *.so* to the piAgent string and load this library, libpiAgent.so. It then calls into this library at a function point defined in the JVMPi specification. The JVM must be able to find this library, so you first need to adjust the LIBPATH environment variable to point to its location (Example 5-33).

Example 5-33 Adjusting LIBPATH to point to libpiAgent.so

```

cd /u/user/hello
export LIBPATH=$LIBPATH:/u/user/AgentController/lib

```

Next, you need to set the CLASSPATH variable so that the profiling agent can find its Java component hcframe.jar as well as the HelloAgain class itself.

Example 5-34 Setting the CLASSPATH variable to locate hcframe.jar

```

export CLASSPATH=$CLASSPATH:./u/user/AgentControllerplugins/org.eclipse.tptp.p
latform.collection.framework_4.1.0/hcframe.jar

```

Now you are ready to launch the program. Make sure that the agent controller is started and then enter the following command (Example 5-35).

Example 5-35 Starting HelloAgain with TPTP profiling enabled

```

java -XrunpiAgent:server=enabled HelloAgain

```

If all is well you should see the output of the program, as shown in Example 5-36, and behind the scenes the profiling agent will create a shared memory connection to the agent controller.


Example 5-36 Example HelloAgain output

```

Hello Profiling World! 1
Hello Profiling World! 2

```

```
Hello Profiling World! 3
Hello Profiling World! 4
Hello Profiling World! 5
Hello Profiling World! 6
Hello Profiling World! 7
Hello Profiling World! 8
Hello Profiling World! 9
Hello Profiling World! 10
Hello Profiling World! 11
....
```

At the Eclipse workbench click the down arrow on the profiling icon . Select the **Profile** option, which will bring up the profile dialog.

On the left side of this dialog double-click **Attach - Java Process**. This will create and select a new profiling configuration of this type.

On the right side of the dialog you are presented with the Host selection tab. Select the z/OS server where that program is running, test the connection, and then click the next tab, Agents.

TPTP will query the Agent Controller on the selected host for its available agents. If the agents list is empty click **Refresh Data**. There should be one agent available; select it and click the > button in the middle of the dialog to move it to the Selected agents list. (Figure 5-37 on page 151).

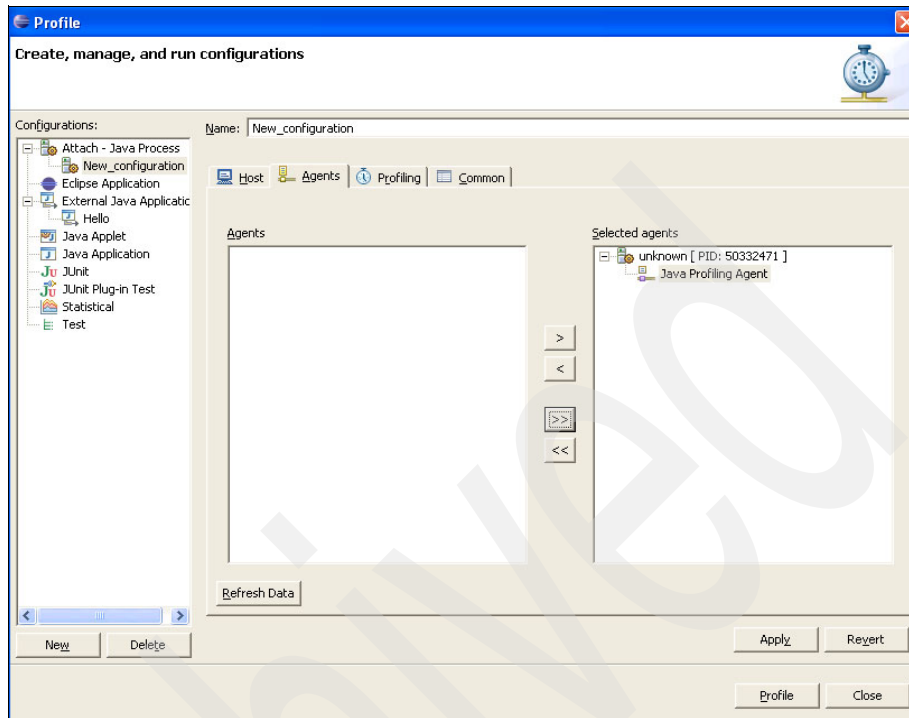


Figure 5-37 Selecting the remote agent

Move on to the Profiling tab, select the Hello profiling set created in the previous example and edit it. Click **Next** until you reach the filter set page and confirm that the **Everything** filter set is selected.

You are now ready to attach to the running process. At the top of the dialog change the name of profile configuration to Hello Again, click **Apply**, and then **Profile**.

You will be returned to the Eclipse workbench and you may be presented with an information dialog stating that although the process is attached at the moment no data is being collected. Click **OK**.

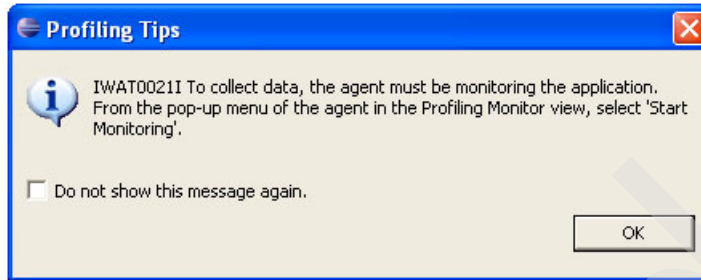


Figure 5-38 The collection warning dialog

In the Profiling Monitor view you will see a new structure representing the attached process. The Agent should be selected; if it is not select it, it is denoted by the text <attached>. Right-click it, and from the context menu select **Open With** and then **Execution flow**. An empty execution flow view will be opened.

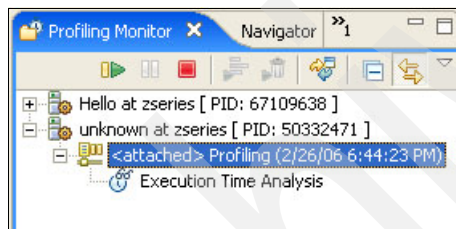



Figure 5-39 The attached, but not collecting, process in the profile monitor

With the agent highlighted, click the **Start Monitoring** button . The workbench will then ask the remote agent controller to instruct the profiling agent to start collecting data. This data is transported via shared memory from the profiling agent to the agent controller, and then over TCP/IP to the Eclipse workbench. The events being received in the messages will be visible at the bottom of the Eclipse window. The profile monitor display will also change to indicate the active collection of data.

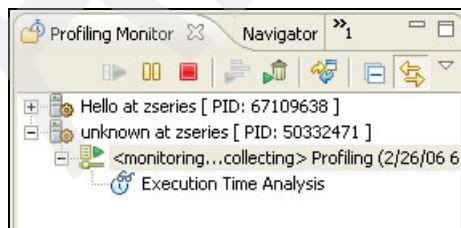


Figure 5-40 The attached and collecting process in the profile monitor

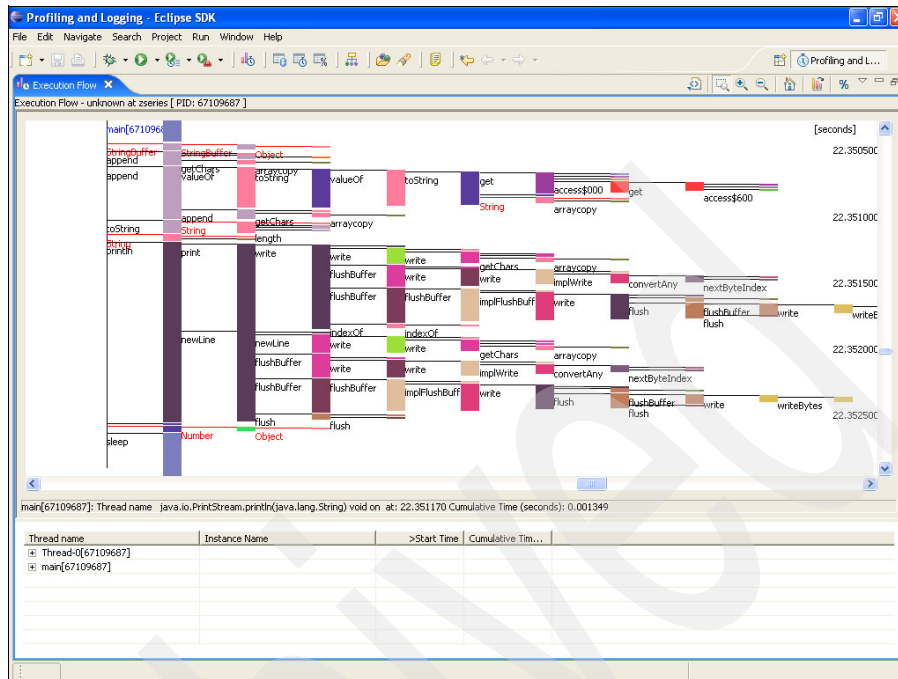


Figure 5-42 The Java activity in between the sleep methods

Here you can see the activity between the sleeps, the construction of the new String from the static **String** and the **Integer.toString**, the **println** and the construction of the new **Integer** from the **Integer.intValue**.

The Execution Flow view, when mastered, is a very powerful weapon in the arsenal of a Java performance analyst. Specifically, it can give an appreciation of the normally hidden cost of vendor and third party libraries. Where more traditional “tree” and “percentage” displays in other tools can leave the user lost after a dozen or so levels, the Execution Flow view remains coherent at any level. It can show both high-level architectural application flow as well as low-level implementation details. It can be used to find bugs and to understand local and remote middleware interactions. Figure 5-43 on page 155 shows the execution of a complex Java program.

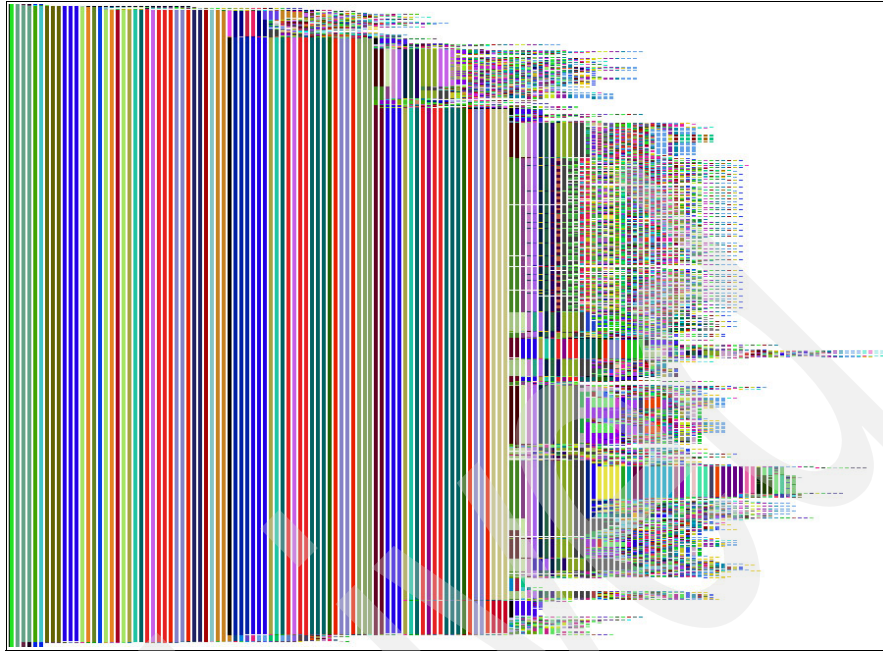


Figure 5-43 An Execution View of a complex Java transaction.

5.6.3 Fractional analysis with TPTP

One of the practical problems with fractional analysis is that it requires the source code of the method one wishes to measure to be changed and recompiled. In many cases the source code for a function may not be available or the rebuild process may be too lengthy or complex.

TPTP, in addition to its profiling tools, provides the ability to change the compiled byte code of a Java class file directly to insert additional arbitrary Java code. This process, known as Byte Code Instrumentation (BCI), makes the instrumentation of third party code very easy.

Wizards assist with the creation of code “fragments,” which can then be inserted into existing class or jar files using easy workbench or command line tools.

At the moment TPTP instrumentation is not reversible, so original copies of the classes or jars files should be kept.

In this example you will be instrumenting, with the **ThreadSafeTimeUsed** method, the JDK **java.lang.Integer.parseInt** method.

Start Eclipse and create a new Java Project. In that project select **File** → **New** → **Other**, scroll down to **Profiling and Logging**, expand its tree and select **Probekit Source File**. (Figure 5-44).

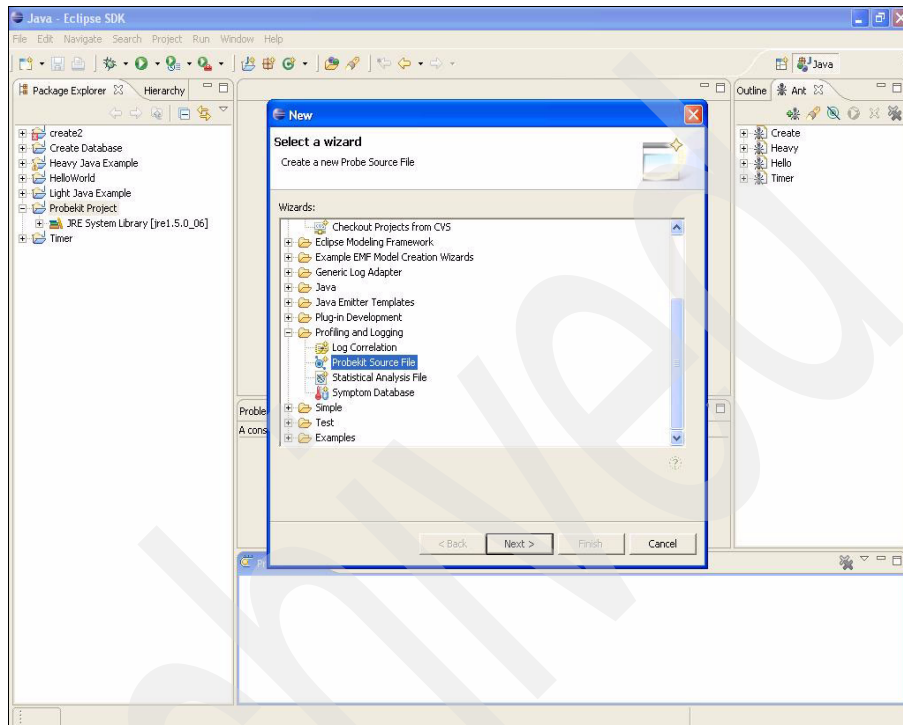


Figure 5-44 Creating a new Probekit source file

A Wizard dialog is opened. Change the name of the probe to `parseInt.probe` and click **Finish** (Figure 5-45 on page 157).

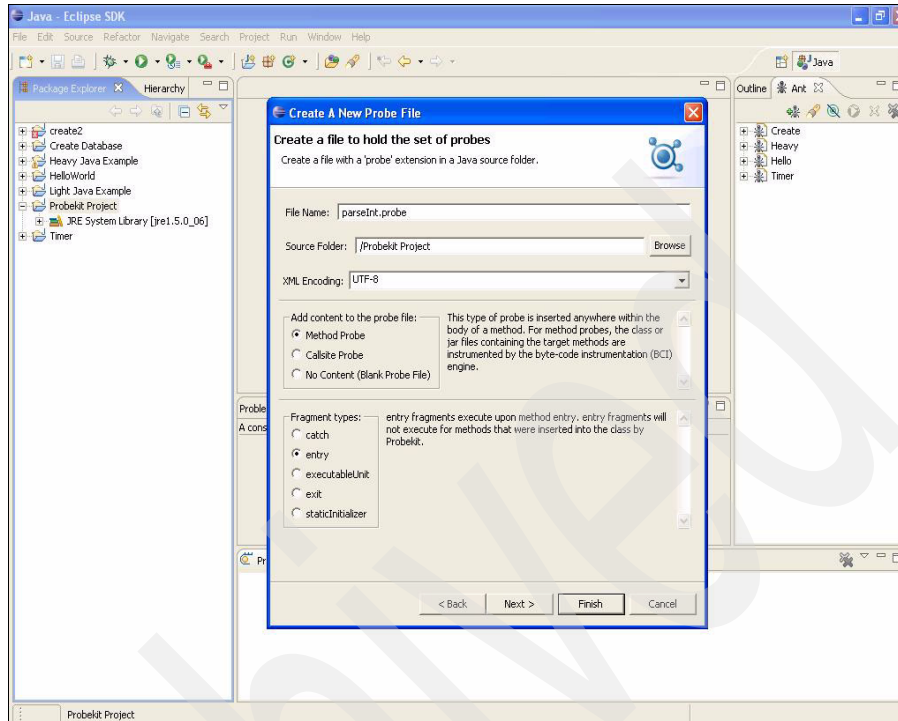


Figure 5-45 Changing the name of the new Probekit in the creation wizard

This will open the new probe file in an editor. Maximize this by double-clicking the tab title.

On the right side of this editor is a tree structure that represents the elements of this probe. Click **Targets** in the tree, then **Add** → **Edit**. Enter the details of the method you wish to instrument using the following procedure.

Set the target to include package `java.lang`, class `Integer`, method **parseInt** and signature `(Ljava/lang/string;)I`.

The method signature is not always required. In the case of **parse int** it has two overloaded methods, normally **parseInt(String)** is called, but this actually just calls **parseInt(String s, int radix)** with a radix, or base, of 10. If you did not specify a signature in your target both versions of the `parseInt` method would be instrumented, although one just immediately calls another, complicating the measurement and decreasing the accuracy.

If you need to specify a method signature, it is important to note that this is defined in a certain “internal” form used by the JVM itself, `(Ljava/lang/string;)I`,

which means one method parameter, an object of type `java.lang.String` and a return value of `int`. This internal format can easily be determined using the `javap` tool included with all JDKs, as shown in Example 5-37.

Example 5-37 Example of the `javap` tool

```
javap -s java.lang.Integer
```

```
....
public static int parseInt(java.lang.String);
    throws java/lang/NumberFormatException
    Signature: (Ljava/lang/String;)I
....
```

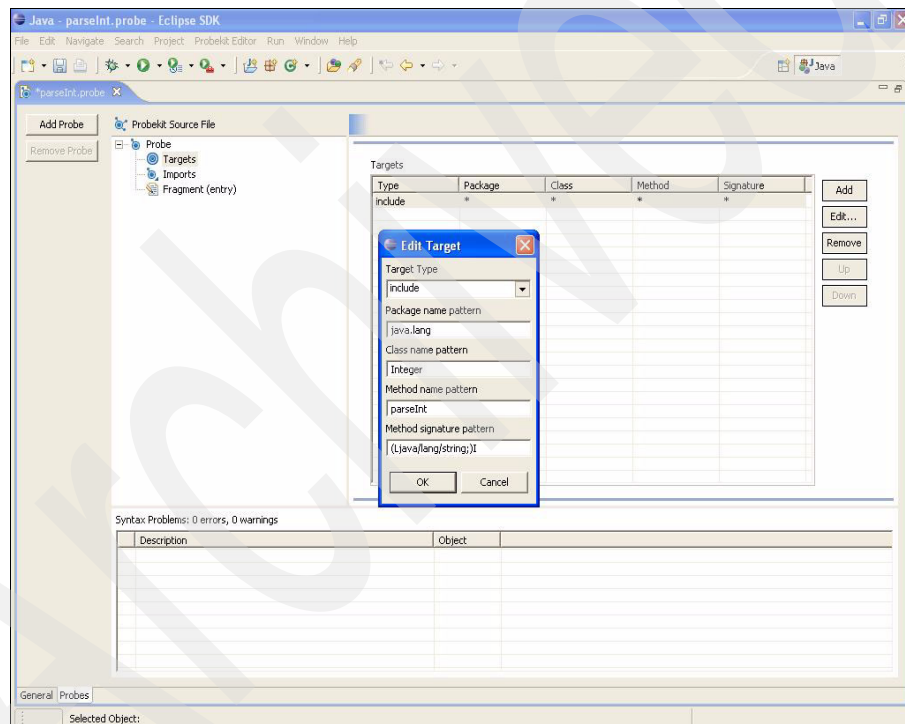


Figure 5-46 Adding the target for the probe

This probe will be used to instrument an individual method, but it could also be applied to other methods, whole classes, or entire packages. Logically, TPTP requires you to add an exclude declaration after your include. Click **Add** and change the type of this new generic entry in the table to **Exclude**.

Because you will be calling the `ThreadSafeTimeUsed` class, you need to add a reference to it in this project. Restore the editor to its original size by double-clicking the tab title and right-clicking the project in the Package Explorer on the right side. Select **Properties** → **Java Build Path**, click the **Projects** tab, then **Add**. Select the `TimeUsed` project that was created in the Fractional Analysis section and click **OK**.

Again, maximize the Probekit editor and select **Imports** in the tree. Here you can add any references to other packages you need to use in your instrumentation code. Click **Add** and input `com.ibm.itso.timer.ThreadSafeTimeUsed` into the new entry field.

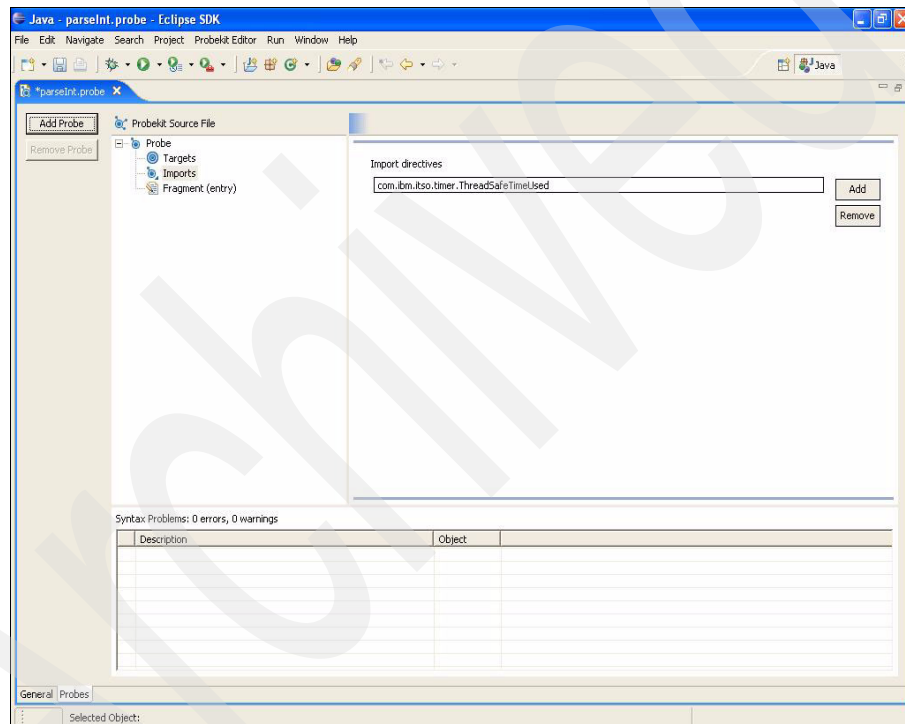


Figure 5-47 Adding an import to the Probekit

Now you can move on to the code fragments you will insert into the `Integer` class file. The wizard has already created a method entry fragment for you, which is in the tree view. Select it and you will see that the Java Code box is empty. In this box enter the code you wish to be executed first whenever this method is called (Example 5-38 on page 160).

Example 5-38 Method entry fragment

```
ThreadSafeTimeused.getTimer("parseInt").start();
```

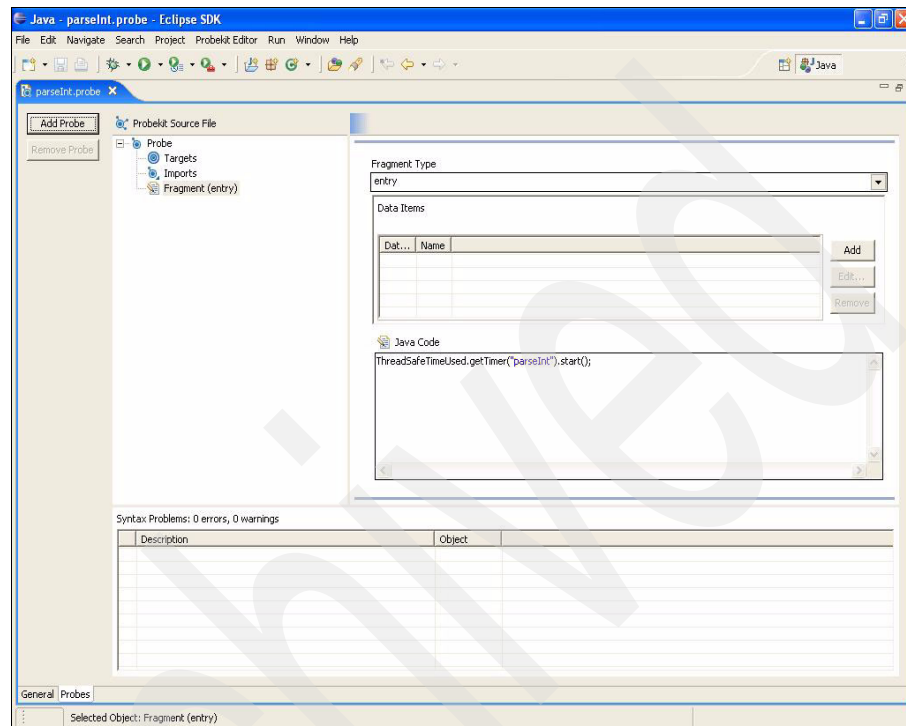


Figure 5-48 Adding code to the method entry fragment

Next, right-click on the top Probe entry in the tree on the left side and select **New** → **Fragment**. This new fragment is added to the bottom of the tree. Click the created fragment, change its type to Exit, and add the code (Example 5-39).

Example 5-39 Method exit fragment

```
ThreadSafeTimeused.getTimer("parseInt").stop();
```

Press Ctrl-S to save the Probe and return to the workbench by double-clicking the Probe editors tab title. Right-click this newly created Probekit source file and select **Probekit** → **Compile**. This creates a new probescript file that will be used to instrument the target jar file, and a java source file that contains the probe itself.

There are two ways to perform the instrumentation, either from the workbench or from the command line. This instrumentation can only be performed on an x86

Linux and Windows. It is more likely when instrumenting for z/OS that this will be done at the command line.

Download core.jar from the JDK lib directory on z/OS. Adjust the windows PATH variable to include a reference to the native TPTP Probekit instrumentation tool, which can be found under the Eclipse root in .\plugins\org.eclipse.hyades.probekit_4.x.x\os\win32\x86, changing 4.x.x to your specific version. Add this as a path entry either permanently in the system settings or temporarily, by opening a command prompt and entering the code shown in Example 5-40.

Example 5-40 Adding the probekit executable to the PATH variable

```
set
PATH=%PATH%;C:\eclipse\plugins\org.eclipse.hyades.probekit_4.1.0\os\win32\x86
```

Change the projects directory under the workspace directory (Example 5-41).

Example 5-41 Changing to the probekit project directory

```
cd C:\workspace\Probekit Project\
```

Add the Probekit jar file to your classpath (Example 5-42).

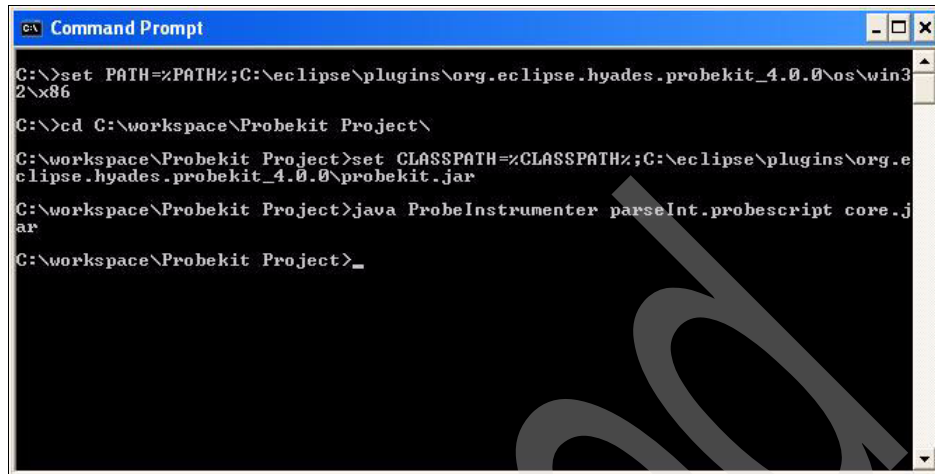
Example 5-42 Adding the Probekit jar file to the classpath

```
set
CLASSPATH=%CLASSPATH%;C:\eclipse\plugins\org.eclipse.hyades.probekit_4.1.0\probekit.jar
```

You are now ready to instrument the JDK's core.jar file and there is already a local copy from the z/OS server in this project directory. The instrumentation is executed as shown in Example 5-43.

Example 5-43 Performing the instrumentation

```
java ProbeInstrumenter parseInt.probescript core.jar
```



```
C:\>set PATH=%PATH%;C:\eclipse\plugins\org.eclipse.hyades.probekit_4.0.0\os\win32\x86
C:\>cd C:\workspace\Probekit Project\
C:\workspace\Probekit Project>set CLASSPATH=%CLASSPATH%;C:\eclipse\plugins\org.eclipse.hyades.probekit_4.0.0\probekit.jar
C:\workspace\Probekit Project>java ProbeInstrumenter parseInt.probescript core.jar
C:\workspace\Probekit Project>_
```

Figure 5-49 Example of an instrumentation session

In the case of an instrumented JDK jar file it would be best to duplicate the directory and point the target application to the instrumented JDK instead of changing the core.jar file in the system JDK.

Although every call to Integer.parseInt will have its CPU consumption measured, there is no call to ThreadSafeTimeUsed.reportAll and you will never get to see the statistics for this timer.

A class that is part of the wider application being measured, or another third party or JDK class could be chosen to insert this reporting probe. Alternatively, a static class level probe could be added to Integer.parseInt to start a thread that reports on a regular basis.

Now that you have your instrumented core.jar file you need to upload it to the z/OS server and place it in your duplicate instrumented JDK directory.

Package the generated probe classes into a jar file, for example probes.jar, upload it to the z/OS server, and add this to the applications classpath (Figure 5-50 on page 163).

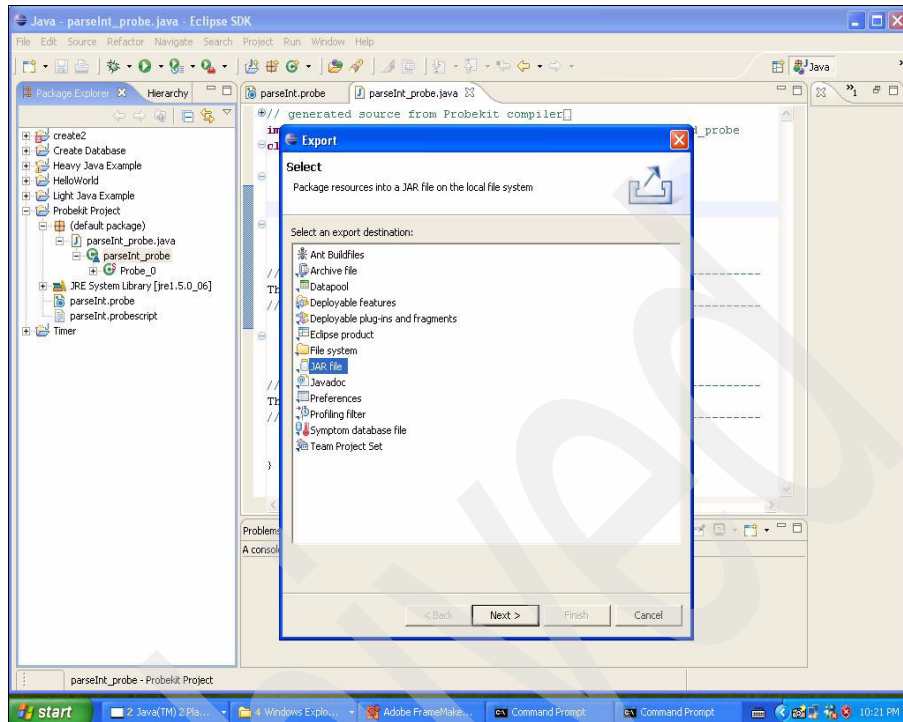


Figure 5-50 Exporting the generated probes as a jar file

Executing a sample application yields the output shown in Example 5-44.

Example 5-44 Example output from the instrumented Integer.parseInt

```

Timer: Integer.parseInt
  Invocations: 1051
  Total CPU Time: 7190us
  Average CPU Time: 43us
  Minimum CPU Time: 8us
  Maximum CPU Time: 22008us
  CPU Overhead: 2 (Calculated but not removed)
  Total Elapsed Time: 63ms
  Average Elapsed Time: 1ms
  Minimum Elapsed Time: 0ms
  Maximum Elapsed Time: 93ms
  Last 5 invocations;
    CPU time: 35us, Elapsed Time: 0ms.
    CPU time: 39us, Elapsed Time: 0ms.
    CPU time: 47us, Elapsed Time: 0ms.
    CPU time: 221us, Elapsed Time: 0ms.
    CPU time: 355us, Elapsed Time: 1ms.

```

Byte Code Instrumentation is a powerful tool and TPTP, with its easy interface, makes it an everyday reality. Although overhead can be calculated and removed, the margin of error when instrumenting an often executed low-level JDK class can produce errant data. Instrumenting a third party library at a higher level is far more practical. Instrumenting a JDK method that the ThreadSafeTimeUsed itself relies on can lead to a failure. For more information about Probekits refer to the online help installed with TPTP.

5.7 Code listings

This section includes the code listings for performance measurement activities referenced in this chapter.

ThreadSafeTimeUsed.java

```
package com.ibm.itso.timer;

import java.io.PrintStream;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Stack;

public class ThreadSafeTimeUsed {

    private static Hashtable timers = new Hashtable();

    private static ThreadLocalTimers threadLocalTimers = new
ThreadLocalTimers();

    public static long overhead = Integer.MAX_VALUE;

    public long invocations = 0;

    public long totalCPU = 0;

    public long totalElapsed = 0;

    public long minimumCPU = Integer.MAX_VALUE;

    public long minimumElapsed = Integer.MAX_VALUE;

    public long maximumCPU = 0;

    public long maximumElapsed = 0;

    public long[] last5 = new long[10];

    public String name = null;

    static {
        System.loadLibrary("timeused");
    }

    static {
        ThreadSafeTimeUsed calibration = null;
        for (int i = 0; i < 5; i++) {
```

```

        calibration = ThreadSafeTimeUsed
            .getTimer("*Internal Calibration Timer " + i);
        for (int j = 0; j < 1000; j++) {
            calibration.start();
            calibration.stop();
        }
        long localOverhead = calibration.totalCPU / calibration.invocations;
        if (localOverhead < overhead ){
            overhead = localOverhead;
        }
        timers.remove("*Internal Calibration Timer " + i);
    }
}

private static class ThreadlocalTimers extends ThreadLocal {

    public Object initialValue() {
        return new Hashtable();
    }

    public Stack getStack(String timerName) {
        Hashtable stacks = (Hashtable) super.get();
        Stack stack = (Stack) stacks.get(timerName);
        if (stack == null) {
            stack = new Stack();
            stacks.put(timerName, stack);
        }
        return stack;
    }
}

private class Timelet {
    long startCPU = 0;
    long startElapsed = 0;

    Timelet(long startCPU, long startElapsed) {
        this.startCPU = startCPU;
        this.startElapsed = startElapsed;
    }
}

public static synchronized ThreadSafeTimeUsed getTimer(String timerName) {
    ThreadSafeTimeUsed timer = (ThreadSafeTimeUsed) timers.get(timerName);
    if (timer == null) {
        timer = new ThreadSafeTimeUsed(timerName);
        timers.put(timerName, timer);
    }
    return timer;
}

```

```

    }

    public ThreadSafeTimeUsed(String name) {
        this.name = name;
    }

    public void start() {
        Stack timeletStack = threadlocalTimers.getStack(this.name);
        timeletStack.push(new Timelet(timeUsed(), System.currentTimeMillis()));
    }

    public void stop() {
        Stack timeletStack = threadlocalTimers.getStack(this.name);
        long stopCPU = timeUsed();
        long stopElapsed = System.currentTimeMillis();
        Timelet timelet = (Timelet) timeletStack.pop();
        long startCPU = timelet.startCPU;
        long startElapsed = timelet.startElapsed;
        long timeCPU = (stopCPU - startCPU);
        long timeElapsed = (stopElapsed - startElapsed);
        synchronized (this) {
            invocations++;

            totalCPU += timeCPU;
            if (timeCPU < minimumCPU) {
                minimumCPU = timeCPU;
            }
            if (timeCPU > maximumCPU) {
                maximumCPU = timeCPU;
            }

            totalElapsed += timeElapsed;
            if (timeElapsed < minimumElapsed) {
                minimumElapsed = timeElapsed;
            }
            if (timeElapsed > maximumElapsed) {
                maximumElapsed = timeElapsed;
            }

            last5[0] = last5[2];
            last5[1] = last5[3];
            last5[2] = last5[4];
            last5[3] = last5[5];
            last5[4] = last5[6];
            last5[5] = last5[7];
            last5[6] = last5[8];
            last5[7] = last5[9];
            last5[8] = timeCPU;
            last5[9] = timeElapsed;
        }
    }

```

```

    }
}

public static void reportAll(PrintStream printStream) {
    Enumeration timerList = timers.elements();
    while (timerList.hasMoreElements()) {
        ((ThreadSafeTimeUsed) timerList.nextElement()).report(printStream);
        printStream.println("");
    }
}

public void report(PrintStream printStream) {
    long averageCPU = 0;
    long averageElapsed = 0;
    if (invocations > 0) {
        averageCPU = totalCPU / invocations;
        averageElapsed = totalElapsed / invocations;
    }

    printStream.println("Timer: " + name);
    printStream.println("  Invocations: " + invocations);
    printStream.println("  Total CPU Time: " + totalCPU + "us");
    printStream.println("  Average CPU Time: " + averageCPU + "us");
    printStream.println("  Minimum CPU Time: " + minimumCPU + "us");
    printStream.println("  Maximum CPU Time: " + maximumCPU + "us");
    printStream.println("  CPU Overhead: " + overhead + " (Calculated but
not removed)");
    printStream.println("  Total Elapsed Time: " + totalElapsed + "ms");
    printStream.println("  Average Elapsed Time: " + averageElapsed + "ms");
    printStream.println("  Minimum Elapsed Time: " + minimumElapsed + "ms");
    printStream.println("  Maximum Elapsed Time: " + maximumElapsed + "ms");
    printStream.println("  Last 5 invocations;");
    for(int i=0; i<5;i++) {
        printStream.println("    CPU time: " + last5[i*2] + "us, Elapsed
Time: " + last5[(i*2)+1] + "ms.");
    }
}

public native static long timeUsed();
}

```

com_ibm_itso_timer_ThreadSafeTimeUsed.h

```

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

```

```

#ifndef _Included_com_ibm_itso_timer_ThreadSafeTimeUsed
#define _Included_com_ibm_itso_timer_ThreadSafeTimeUsed
#ifdef __cplusplus
extern "C" {
#endif
/* Inaccessible static: timers */
/* Inaccessible static: threadlocalTimers */
/* Inaccessible static: overhead */
/*
 * Class:      com_ibm_itso_timer_ThreadSafeTimeUsed
 * Method:     timeUsed
 * Signature:  ()J
 */
JNIEXPORT jlong JNICALL Java_com_ibm_itso_timer_ThreadSafeTimeUsed_timeUsed
    (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif

```

com_ibm_itso_timer_ThreadSafeTimeUsed.h

```

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

#ifndef _Included_com_ibm_itso_timer_ThreadSafeTimeUsed
#define _Included_com_ibm_itso_timer_ThreadSafeTimeUsed
#ifdef __cplusplus
extern "C" {
#endif
/* Inaccessible static: timers */
/* Inaccessible static: threadlocalTimers */
/* Inaccessible static: overhead */
/*
 * Class:      com_ibm_itso_timer_ThreadSafeTimeUsed
 * Method:     timeUsed
 * Signature:  ()J
 */
JNIEXPORT jlong JNICALL Java_com_ibm_itso_timer_ThreadSafeTimeUsed_timeUsed
    (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif

```

com_ibm_itso_timer_ThreadSafeTimeUsed.c

```
#include "com_ibm_itso_timer_ThreadSafeTimeUsed.h"

void CPU (void *);
#pragma linkage(CPU,OS)

JNIEXPORT jlong JNICALL Java_com_ibm_itso_timer_ThreadSafeTimeUsed_timeUsed
    (JNIEnv * env, jobject thisObject) {
    jlong microseconds = 0L;

    CPU(&microseconds);

    return microseconds;
}
```

cpu.s

```
*      PRINT NOGEN
R1      EQU 1
R2      EQU 2
R3      EQU 3
R4      EQU 4
R5      EQU 5
R6      EQU 6
R7      EQU 7
R8      EQU 8
R9      EQU 9
R10     EQU 10
R11     EQU 11
R12     EQU 12
R13     EQU 13
R14     EQU 14
R15     EQU 15
CPU_PARMS DSECT
CPU_PARMS_START DS OF
CPU_PARMS_FIRST_PTR DS F
CPU      RMODE ANY
CPU      AMODE 31
CPU      CSECT
        USING CPU,R15
        B      DOIT
        DC      CL8'CPU--STR'
DOIT     DS      0H
        SAVE   (14,12)
        LR     R10,R15
HERE     DS      0H
        L      R2,0(,R1)
```

```
TIMEUSED STORADR=(2),LINKAGE=SYSTEM,CPU=MIC  
RETURN (14,12),RC=0  
DC CL8'CPU--END'  
END
```

Build commands

```
javac com/ibm/itso/timer/ThreadSafeTimeUsed
```

```
javah com.ibm.itso.timer.ThreadSafeTimeUsed
```

```
c89 -o libtimeused.so -W "l,xplink,dll" -W  
"c,langl1l(extended),xplink,dll,exportall" -I/usr/lpp/java/J1.4/include  
com_ibm_itso_timer_ThreadSafeTimeUsed.c cpu.s  
/usr/lpp/java/J1.4/bin/classic/libjvm.x
```

Exploiting zAAP

In this chapter, we provide an overview of IBM eServer System z Application Assist Processor (zAAP) and present experimental results for potential zAAP utilization of various APIs that are typically used in Java batch applications. This includes pure Java calculations including bitwise operations, Java Record I/O library (JRIO), and JDBC drivers for DB2 on z/OS.

The idea behind these experiments is to provide an estimate of potential zAAP utilization of various applications at much finer granularity than previously published estimates offered. The potential benefits of zAAP can be more clearly evaluated by considering and estimating the type of APIs that are used in a new or existing mainframe application.

Some of the experimental results look promising, especially the results of experiments with the JRIO library. Our conclusion, based on the observation of the experimental results, is that zAAP provides good potential for Java batch applications in terms of managing the total cost of mainframe-based applications. The results indicate that the Java language is a good candidate to develop new batch applications as well as replacing existing batch applications in certain situations.

6.1 zAAP prerequisites

The zAAP is not available for all machine types and environments. There are three prerequisites:

- ▶ IBM System z9 or IBM eServer™ zSeries 990 (z990) or zSeries 890 (z890) system
- ▶ z/OS 1.6 or higher
- ▶ IBM SDK for z/OS, Java 2 Technology Edition, 5655-I56, and later z/OS SDK products

For additional information refer to:

<http://www-03.ibm.com/systems/z/zaap/>

6.2 Overview of zAAP and runtime options

IBM eServer System z Application Assist Processor (zAAP) is an attractively priced specialized processing unit that provides an economical Java execution environment for customers who desire the traditional qualities of service and integration advantages of the zSeries platform. zAAP processors alone cannot perform any tasks, unlike other processing units. They can only be configured to assist the general purpose CPs in the execution of Java applications. For this reason, IBM does not impose software charges on zAAP capacity.

Once zAAPs are installed, the IBM JVM processing cycles can be executed on the configured zAAPs with no modifications to the Java applications. zAAPs can help increase the general purpose processor productivity of a system and may contribute to lowering the overall cost of computing using Java applications on zSeries. The amount of general purpose processor savings will vary based on the amount of Java application code executed by zAAPs.

The following runtime options provided for the IBM JVM are related to zAAP:

-Xifa:on	Enables Java work to run on a zAAP if any zAAPs are available. This is the default option.
-Xifa:off	Disables the usage of zAAPs.
-Xifa:projectn	Estimates projected zAAP usage and writes this information to STDOUT at intervals of <i>n</i> minutes.
-xifa:force	Forces Java to continue attempting to use zAAPs, even if none are available.

In addition to these JVM options, there are two options that can be configured to control how zAAP-eligible work is executed on the system:

IFACrossOver=YES	zAAP-eligible work can run on both zAAPs and standard CP processors.
IFACrossOver=NO	zAAP-eligible work can only run on zAAP unless there are no operational zAAPs installed.
IFAHonorPriority=YES	WLM manages the priority of zAAP-eligible work for CPs. zAAP-eligible work is handled by WLM in priority order.
IFAHonorPriority=NO	zAAP-eligible work can run on CPs but at a priority lower than any non-zAAP work.

6.3 Exploring zAAP utilization potentials

In the following sections we discuss how zAAP utilization can be estimated and measured.

6.3.1 zAAP utilization estimate and projection tool

Before making the decision to purchase and install zAAPs, potential zAAP utilization of Java applications should be considered to maximize the return on investment. One extreme situation is where 99% of the application is written natively and the application is invoked via the Java Native Interface (JNI). For such applications, the benefit of zAAPs is very limited (zAAPs are not utilized at all). Another extreme situation is where most of the application is written purely in Java. In this case one could expect a very high zAAP utilization to run the application. The estimate of zAAP utilization is usually not so straightforward because typical applications fall anywhere between the two extremes just mentioned.

IBM has measured several known workloads in order to estimate the ratio of Java execution time to the total system CPU time. Table 6-1 summarizes different types of Java applications and their estimated zAAP utilization potentials. It gives a very high-level overview of zAAP utilization potentials for typical enterprise scenarios. As we expected, it shows a wide range of potential zAAP utilizations depending on the constitution of applications, and it can be influenced by many factors including processor configurations, software levels including z/OS, WebSphere, and the SDK.

Table 6-1 Java content from IBM measured workloads

Workload name	Java % of total CPU time	Description of workload
XML Parse	98%	A CPU-intensive XML parsing workload that repeatedly parses XML documents using either SAX or DOM APIs, both with and without validity checking.
Trade 2	40%	IBM-developed workload modeling an electronic brokerage providing online securities trading. Provides a real-world eBusiness application mix of HTTP sessions, servlets, JSPs, stateless session EJBs, container-managed persistent EJBs (CMPs), and JDBC data access to DB2. Generated with VA Java tooling, and characterized as light SQL with a small database component.
Trade 3	60%	Third generation of WebSphere end-to-end benchmark evolved from Trade 2 and covering J2EE 1.3, including the EJB 2.0 component architecture, message-driven beans with PUB/SUB and point-to-point asynchronous messaging. Characterized as light SQL with a small database component.
CTG/CICS/ERWW	40%	Web-enabled access to a traditional CICS/DB2 OLTP database environment. J2EE application for legacy CICS transactions using servlets, JSPs, stateless session EJBs and the CICS Transaction Gateway (CTG). The size of the messages being passed between WebSphere and CICS is relatively small. All the business logic is in the legacy CICS transactions and not in Java.
IMS Connect ERWW	40%	Web-enabled access to a traditional IMS/DB2 OLTP database environment. J2EE application for legacy IMS transactions using servlets, JSPs, stateless session EJBs and IMS Connect. The size of the messages being passed between WebSphere and IMS is relatively small. All the business logic is in the legacy IMS transactions and not in Java.
OLTP-T	0%	Traditional IMS LSPR online workload. Consists of light to moderate IMS transactions from DLI applications and using SNA LU2 message streams.

Although such estimates are helpful in getting a high-level overview of zAAP utilization potentials of various enterprise applications that consist of multiple products, they do not provide any indication about individual product or individual API utilization potentials.

To get a better understanding of zAAP potential utilization of individual products and APIs, we developed a set of Java programs that perform a very specific objective to study the potential zAAP utilization. For example, we have a simple Java program that performs I/O operations to a VSAM data set indefinitely. By running this program and using the utilization projection tool of IBM SDK 1.4, we measured the potential zAAP utilization of applications using the JRIO library. To get the potential zAAP utilization estimates using the projection tool, the Java application should be started as follows:

```
java -Xifa:projectn MyJavaAppName
```

The -Xifa:project option turns the projection tool on for the specified Java application, and *n* specifies the interval (in minutes) that the potential zAAP utilization is to be captured from the running Java application. The projection tool prints out the utilization information to the standard error output stream as follows:

```
IFA Projection data for system id=<SC76.16777265> Starting at: 13:01:30 - Current address space
CPU: 0.035453 sec.
<SC76.16777265> Interval at: 13:02:31 Switches To/From IFA: 1309 Java IFA: 5.327046 sec. Java
Standard CPU 0.355675 sec. Total address space CPU: 5.720031 sec.
<SC76.16777265> Interval at: 13:03:32 Switches To/From IFA: 1316 Java IFA: 4.645807 sec. Java
Standard CPU 0.362515 sec. Total address space CPU: 5.043859 sec.
<SC76.16777265> Interval at: 13:04:33 Switches To/From IFA: 1328 Java IFA: 4.228194 sec. Java
Standard CPU 0.364403 sec. Total address space CPU: 4.627972 sec.
<SC76.16777265> Interval at: 13:05:34 Switches To/From IFA: 1326 Java IFA: 4.228403 sec. Java
Standard CPU 0.366141 sec. Total address space CPU: 4.630676 sec.
<SC76.16777265> Interval at: 13:06:35 Switches To/From IFA: 1326 Java IFA: 4.240411 sec. Java
Standard CPU 0.384171 sec. Total address space CPU: 4.661109 sec.
<SC76.16777265> Interval at: 13:06:42 Switches To/From IFA: 165 Java IFA: 0.502153 sec. Java
Standard CPU 0.043451 sec. Total address space CPU: 0.550255 sec.
<SC76.16777265> TOTAL at: 13:06:42 Switches To/From IFA: 6770 Java IFA: 23.172014 sec. Java
Standard CPU 1.876356 sec. Total address space CPU: 25.233902 sec.
```

A Microsoft® Excel® workbook is available that processes the output lines and prepares summary information in a spreadsheet. The tool is available from:

<https://www6.software.ibm.com/dl/zosjava2/zosjava2-p>

Figure 6-1 shows a portion of the output from the MicroSoft Excel zAAP utilization estimate workbook that is based on the output of the projection tool shown previously. In this particular case, it indicates a very high level of zAAP utilization potential (above 90%).

day/time at start of interval	zAAP eligible seconds	Java not eligible seconds	Space CPU seconds	%Time zAAP eligible	duration
z900	Go to Inventory		Service Class		
1 13:01:30	23	2	25	92%	360
1 13:01:31	5	0	6	93%	60
1 13:02:32	5	0	5	92%	60
1 13:03:33	4	0	5	91%	60
1 13:04:34	4	0	5	91%	60
1 13:05:35	4	0	5	91%	60
1 13:05:42	1	0	1	91%	60

Figure 6-1 Output of zAAP utilization projection tool

6.3.2 Experimental results of zAAP utilization estimates

In this exploration, we studied the following scenarios:

1. Type conversions using JRecord
2. HFS files access with Java I/O library
3. HFS files access with Java Native Interface (JNI)
4. HFS files access with Java I/O library and JNI
5. VSAM data set access with JRIO Library
6. DB2 read access with Type 2 driver
7. DB2 read access with Type 4 driver
8. DB2 update access with Type 2 driver
9. DB2 update access with Type 4 driver

Table 6-2 summarizes the results. Details about each experiment and our observations are presented following the table.

Table 6-2 Results of potential zAAP utilization estimates

Scenarios	Estimated zAAP utilization (%)
1 (Conversions from/to COBOL type)	100%
2 (Java I/O library)	100%
3 (C I/O using JNI)	0%
4 (Java I/O + JNI)	49%
5 (VSAM with JRIO)	100%
6 (DB2 read with Type 2 driver)	45%
7 (DB2 read with Type 4 driver)	100%
8 (DB2 update with Type 2 driver)	7%
9 (DB2 update with Type 4 driver)	100%

Scenario 1: Type conversions using JRecord

The JRecord library described in 3.4, “JRecord bean generator and supporting APIs” on page 50 provides utility methods to convert an array of bytes to various Java types such as integers and Strings. The conversion methods to and from byte arrays to Java integers perform numerical calculations and various bitwise operations using shift, AND, and OR operators. In this experiment, we used a simple Java program that performs conversion of Java integers to their corresponding COBOL packed decimal formats, then back to the original format using the JRecord library.

Since all methods in the library are implemented purely in Java, we expected to see a full utilization of zAAP from the experiment. The result confirmed our expectation and showed a potential of 100% utilization of zAAP.

Scenario 2: HFS files access with Java I/O library

In this experiment, we used a simple Java program to create a copy of a file in the HFS given a name of the file in HFS. The program uses classes from the Java I/O library, including FileInputStream and FileOutputStream, to implement the experimental program.

We expected to see 100% potential utilization of zAAP in this experiment and the result confirmed our expectation.

Scenario 3: HFS files access with Java Native Interface (JNI)

In this experiment, we used a program that performs the same functionality as the previous experiment (HFS file copy with Java I/O library) except the HFS file

I/O operations are implemented in a C routine and the routine is called from the Java program using the JNI interface.

Because most of the functionality is implemented in the C routine, we expected close to zero potential utilization of zAAP in this experiment. The result confirmed our expectation, showing 0% potential utilization of zAAP once the program is up and running.

Scenario 4: HFS files access with Java I/O library and JNI

In this experiment, we used a Java program that combines the previous two experiments. That is, the program copies a file in the HFS using the Java I/O library and JNI interface interchangeably. This experiment was intended as a verification process of the accuracy of the projection tool distributed with IBM SDK 1.4.

We expected to achieve 50% potential utilization of zAAP in this experiment because HFS files are accessed using the Java I/O library and JNI interface interchangeably. The result confirmed our expectation (49% potential utilization) and verified the validity of the projection tool.

Scenario 5: VSAM data set access with JRIO library

In this experiment, we used a simple Java application that reads and updates records from a VSAM data set using the JRIO library. The JRIO library is a fully zAAP-enabled official API to access data sets and HFS files. Therefore, we expected close to a full utilization of zAAP in this experiment.

The results from the experiment exceeded our initial expectations and showed full utilization of zAAP once the program is up and running in this experiment.

This is an interesting result in that it gives incentives to develop new batch Java applications or re-write existing batch applications using Java if such applications are concerned with I/O processing of data sets. By using the JRIO library, the new batch jobs would run completely on zAAP processors, which potentially gives several incentives. This approach:

- ▶ Frees the usage of main processing units for other tasks, taming the overall cost of ownership of mainframe applications.
- ▶ Takes advantage of competitive price offerings of zAAP processors.
- ▶ Enables modernization of existing applications by re-implementing such applications in Java.
- ▶ Becomes accessible because it is easier to find Java developers to manage mainframe applications.

This is one of the key advantages of using the JRIO library rather than developing a new customized version of APIs to access data sets using JNI interfaces. As we described earlier, JNI-based solutions would not utilize zAAPs.

Scenarios 6 through 9: DB2 access with different drivers

The JDBC specification describes four different types of driver architectures to connect Java applications to databases. Among the four architectures, Type 2 and Type 4 drivers are the most commonly used in enterprise applications today. Type 2 drivers are written partly in the Java programming language and the rest is written in native code. The drivers use a native client library specific to the data source to which they connect. Type 4 drivers are written purely in Java and implement the network protocol for a specific data source.

In this experiment, we developed a pair of Java programs for each driver type to get the estimate of potential zAAP utilization. The first program reads data stored in an existing database table using the SELECT operation. The second program deletes a set of tables, re-creates the table and populates the table with randomly generated data. The same set of programs is used every time, except for the driver type used for each experiment.

Because of the architectural specification of the drivers, we expected almost full zAAP utilization with the Type 4 drivers and much less utilization with the Type 2 drivers.

The results confirmed our expectations of the potential utilization of zAAP. When we used the Type 4 drivers, we got 100% utilization of zAAP once the programs were up and running in both scenarios¹. The Type 2 driver experiments also showed the expected results: 45% utilization for the read database access and 7% for the update database access.

6.3.3 Hardware and software configuration

We used the following hardware and software for our experiments:

- ▶ Hardware configuration:
 - z9 LPAR with 2 general-purpose processors and 2 zAAPs installed with 4GB of online real storage
- ▶ Software configuration:
 - z/OS release 1.7
 - IBM SDK for z/OS, Java 2 Technology Edition, V1.4.2
 - Language Environment® at z/OS release 1.7 level
 - DB2 Version 8

¹ Although our experiments show a full utilization potential of zAAP with the Type 4 driver, the overhead of the network protocol layer on DB2 must be considered because the projection tool does not include the estimate of such additional overhead.

Problem determination

This chapter describes how to determine the cause of problems encountered with stand-alone Java applications on z/OS.

The following topics are included:

- ▶ Problems with the z/OS environment
- ▶ Collecting data for problem determination
- ▶ Dump analysis and tools for problem determination
- ▶ Problem determination scenarios

Additional information is in the Diagnostic Guide at:

<http://www-128.ibm.com/developerworks/java/jdk/diagnosis/>

7.1 Introduction

The goal of problem determination, in its most basic sense, is to get to the root of a problem. It is similar to what a programmer might call debugging, but on a more comprehensive scale. The problem determination discussed in this chapter refers to finding the root of a problem that could not be solved by using a debugger in an IDE.

Sometimes, using a debugger is not even possible, for instance in a production environment. To analyze heap-related issues (such as `OutOfMemoryError` and other crashes, and hangs or loops within your application), with a level of detail similar to that of the traditional COBOL environment, the relatively low impact tool of unformatted dumps and JVM internal information should be used.

Whenever a Java stand-alone application running on z/OS has a runtime problem, one normally checks the following areas:

- ▶ The z/OS environment
- ▶ Any output/log from the application itself
- ▶ Different types of dumps from the JVM process

Checking the z/OS environment and analyzing JVM dumps are always very challenging to application development people.

The discussions in this chapter apply to the IBM SDK for z/OS Java 2 Technology Edition, V1.4 and later. The operating system used in the discussions is z/OS version 1.7, but almost all discussions apply to older versions of z/OS as well.

7.2 Checking the z/OS environment

A good understanding of the runtime environment for the application is very important for problem determination. Since z/OS is such a complicated environment, application development/support people should work with z/OS system programmers to check the application runtime environment.

7.2.1 Maintenance

Check and document the maintenance level for:

- ▶ z/OS operating system
- ▶ Application-related subsystems such as DB2, WebSphere MQ, CICS, and so on

- ▶ Application-related subsystem connectors
- ▶ XML toolkit or parser in use on z/OS
- ▶ Java for z/OS

The most up-to-date information about operating system prerequisites for correct JVM operation is found at IBM's Java for z/OS Web site:

<http://www.ibm.com/servers/eserver/zseries/software/java/>

In addition, any new prerequisites are described in PTF HOLDDATA.

Keeping and tracing this information can be very helpful to determine if the problem is caused by wrong maintenance levels or mismatched maintenance levels. Sometimes the maintenance level changes can make your application malfunction.

If you talk to the IBM support team, they will ask you to provide this basic information so that they can assist you in investigating the problems.

7.2.2 LE settings

The JVM running on z/OS is written in C/C++. The z/OS Language Environment (LE) Runtime Options (RTOs) affect operation of C and C++ programs.

In general, you should not use your own LE RTOs to override the options that were set by the JVM developers using C #pragma statements in the code. These options are generated as a result of extensive testing to provide the best possible operation of the JVM.

If you do need to override some of the LE RTOs for some reason, we recommend you do some testing to evaluate the impact on the JVM and document the changes for future problem determination.

7.2.3 Environment variables

Environment variables that change the operation of the IBM JVM in one release can be deprecated or change meaning in a following release. Therefore, you should review environment variables that are set for a specific release of the JVM to verify that they still apply after a future upgrade.

Any JVM or other software upgrade can make your environment variables such as PATH, LIBPATH, CLASSPATH, and so forth no longer valid. You need to double check the USS environment variables in your USS shell scripts and JCLs to ensure they are still valid.

To make life easier, we recommend that system programmers create a set of symbolic links in USS. Application developers only use the symbolic names in their shell scripts and JCLs, and system programmers could take the responsibility to keep the links up-to-date.

7.2.4 Private storage usage

The most common causes of failure after a successful installation of IBM JVM for z/OS are those related to insufficient private storage. In Java developer's words, the problem is that there is not enough memory to run the JVM.

The LE HEAP is used for the allocation of the Java heap where instances of Java objects are managed by garbage collections. For example, the JIT compiler obtains work areas for compilation of methods and to store compiled code.

Since the JVM must preallocate the maximum Java heap size so that it is contiguous, the total private storage (memory) needed to run a JVM is that of the maximum Java heap size, plus an allowance for underlying allocations.

To allow a JVM having the default maximum heap size (64MB), the private storage should be 140 MB. If the private storage is restricted by a system parameter or user exit, private storage allocation failures occur. These failures show as `OutOfMemoryErrors` or `Exceptions`, failures to load DLLs, or failures to complete subcomponent initialization during startup.

If you log in from TSO/E, and cannot start the JVM successfully because of memory issues, you should check the Size value on the TSO/E logon panel. Increase the Size number and then try to start the JVM again. The max Size value is 2096128.

7.2.5 Service Class for the application

Workloads on z/OS are all controlled by the z/OS Work Load Manager (zWLM).

zWLM classifies workloads into different categories with different performance goals and importances. Each workload category consists of workloads having the same performance goal and importance level. The workload category is defined in zWLM as a *Service Class*.

zWLM manages the computing resources and workloads on the System z servers to make sure the workloads meet the performance goals defined in their Service Classes. If there are not enough resources available on the system to allow the workloads to meet their performance goals, zWLM favors the workload having a higher importance level.

If a stand-alone Java application is launched from USS using a shell script, it runs in a USS address space.

If it is launched from a JCL using the BPXBATCH utility, there are three address spaces involved to run the Java program. The first one is the job running BPXBATCH, the second one is the USS session created by BPXBATCH, and the third is a USS address space in which the JVM is launched and in which the Java program runs. By default, fork and spawn set jobname values to the user ID with a number (1-9) appended.

No matter whether it is launched using a shell script or the BPXBATCH utility, since the Java program runs in a USS address space, the workload should be classified into one of the Service Classes defined for the OMVS subsystem. Note that the Service Class under which the Java program runs can be different from that of the BPXBATCH utility job.

If the Java program is launched using a custom JVM launcher, there might be only one address space involved. In this case the Service Class under which the job is running governs the performance goal and importance of the Java program.

Understanding the Service Class under which your Java program is running and its relationship to the Service Classes of other subsystems is very useful to determine some performance-related problems.

7.3 Collecting data for problem determination

The data collected from a fault situation of a Java stand-alone application on z/OS depends on the problem symptoms, but could include some or all the followings:

- ▶ CEEDUMP
- ▶ SYSTDUMP
- ▶ SVCDUMP
- ▶ JAVADUMP
- ▶ HEAPDUMP
- ▶ Binary or formatted trace data from JVM internal high performance trace
- ▶ Debugging messages written to stderr by JVM
- ▶ Trace of debugging message from your Java application

The following sections describe how to collect and use this data for problem determination.

7.3.1 JVM dump control

The IBM JVM supports a simple scripting ability to choose when and how a dump is generated.

A dump might be produced in response to specific events, depending on the setting of the environment variables `JAVA_DUMP_OPTS` and `JAVA_DUMP_TOOL`, as well as any `-Xdump` agent options specified on the command line.

On z/OS, the full syntax for `JAVA_DUMP_OPTS` is:

```
JAVA_DUMP_OPTS="ONcondition(dumptype,dumptype),ONcondition  
(dumptype,...),...,USERABEND(nnnn),ceedumpoptions"
```

`JAVA_DUMP_OPTS` is parsed by taking the first (leftmost) occurrence of each condition, so duplicates are ignored.

On z/OS, the valid dump types are:

- ▶ ALL
- ▶ NONE
- ▶ JAVADUMP
- ▶ SYSDUMP
- ▶ CEEDUMP
- ▶ HEAPDUMP

If `USERABEND` is set, it must specify an integer 1 through 4094, or it is ignored. If set, it takes precedence over all other settings and for any condition, the LE terminates the JVM with the specified abend code, and bypasses any cleanup routines. The abend code can be intercepted by using a SLIP TRAP to take a SYSDUMP.

If `CEEDUMP` is specified, an LE `CEEDUMP` is produced for the relevant conditions, after any `SYSDUMP` processing, but before a `JAVADUMP` is produced. The default options for `CEEDUMP` are:

```
"THREAD(ALL),PAGESIZE(0),ENC(CUR),NOENTRY,GEN0"
```

Five categories of events can trigger producing a dump:

- ▶ **EXCEPTION**, which is an unexpected synchronous terminating signal; it is an unrecoverable storage violation. On z/OS it corresponds to the USS signal `SIGTRAP`, `SIGILL`, `SIGSEGV`, `SIGFPE`, `SIGBUS`, `SIGSYS`, `SIGXCPU`, `SIGXFSZ`.
- ▶ **ERROR**, which is a controlled abort because an error is detected internally, such as no more storage available. It corresponds to the USS signal `SIGABRT`.

- ▶ INTERRUPT, which is an asynchronous terminating signal, for example if you press CTRL-C. It corresponds to the USS signal SIGINT, SIGTERM, SIGHUP.
- ▶ DUMP, which can be caused by pressing CTRL-V on z/OS. It corresponds to the USS signal SIGQUIT.
- ▶ OUTFMEMORY

Using JAVA_DUMP_OPTS you can specify which type of dump is taken under which condition. The default value of the JAVA_DUMP_OPTS on z/OS is:

```
JAVA_DUMP_OPTS="ONANY SIGNAL(JAVADUMP,SYSDUMP),ONINTERRUPT(NONE)"
```

The JAVA_DUMP_TOOL environment variable allows the users to using their own tools to take dump. The tool specified by JAVA_DUMP_TOOL is run after any system and heap dump has been taken, but before anything else.

Dumps are generated in the following forms:

- ▶ SYSDUMP
A standard MVS data set with the default name in the form of %uid.JVM.TDUMP.%job.D%Y%m%d.T%H%M%S, or you can specify your own name by setting the JAVA_DUMP_TDUMP_PATTERN environment variable.
- ▶ CEEDUMP
You can either find this dump in the current HFS directory or specify your own place by setting the _CEE_DMPTARG environment variable. The file name is CEEDUMP.%Y%m%d.T%H%M%S.%pid.
- ▶ HEAPDUMP
In the same HFS directory with the CEEDUMP, a file with the name HEAPDUMP.%Y%m%d.%H%M%S.%pid.phd, or as determined by the JAVA_DUMP_HDUMP_PATTERN environment variable.
- ▶ JAVADUMP
In the same HFS directory with the CEEDUMP, or standard JAVADUMP directory as: JAVADUMP.%Y%m%d.%H%M%S.txt.

Refer to *IBM JDK 1.4.2 Diagnostics Guide*, SC34-6359, for more information about how to use dump agents.

7.3.2 CEEDUMP, SYSTDUMP and SVCDUMP

LE CEEDUMP is a formatted application-level dump, requested by the **cdump** system call. CEEDUMP shows stack traces for each thread that is in the JVM process, together with register information and a short dump of storage for each register.

To capture an LE CEEDUMP, you have to specify CEEDUMP in the environment variable JAVA_DUMP_OPTS.

Since CEEDUMP is a formatted dump, it is readable without using any special tool. In each case, the dump shows the C-stack, which is separate from the Java stack that is built by the JVM because one method gives control to another. The C-stack frames are also known as DSAs on z/OS. It is the name of the control block the LE provides as a native stack frame for a C/C++ program.

Under z/OS, the behavior of LE can be changed by setting the _CEE_RUNOPTS environment variable. The TRAP option determines whether LE condition handling is enabled, which, in turn, drives JVM signal handling. The TERMTHDACT option indicates the level of diagnostic information that LE should produce. Refer to the z/OS language environment document *z/OS Language Environment Debugging Guide*, GA22-7560 for information on how to read/analyze the CEEDUMP.

System Transaction Dump (SYSTDUMP) is an unformatted dump created by the BCP IEATDUMP service.

The IBM JDK 1.4.2 JVM, by default, generates a SYSTDUMP when any of the following happens:

- ▶ A SIGQUIT signal is received (using `kill -3 pid`)
- ▶ The JVM aborts because of a fatal error
- ▶ An unexpected native exception occurs (for example, a SIGSEV, SIGILL, or SIGFPE is received)

SYSTDUMP is needed when CEEDUMP does not provide enough information for problem diagnosis due to the limited data captured. It contains information about both native and Java threads, stacks, locks, traces, storage, GC, and so forth. Since SYSTDUMP is unformatted, it needs to be analyzed via tools such as IPCS, jformat, or svcdump.jar.

To view the CEEDUMP contents within the SYSTDUMP, use the Interactive Problem Control System (IPCS) verbexit LEDATA, with the CEEDUMP option to format and analyze LE control blocks.

SVCDUMP is a core dump initiated by the operating system, generally when a programming exception occurs. SVC dump processing stores data in MVS dump data sets preallocated or automatically allocated as needed.

SVCDUMP can also be initiated from the MVS console to gather diagnostic data for a “hang” condition. An SVCDUMP captured by this way is called a console dump.

SVCDUMP captures even more data than SYSDUMP, and it is unformatted as well. To analyze SVCDUMP, you can use the same tools used for SYSDUMP analysis. For example, jformat, IPCS, and svcdump.jar can be used to extract information you find in CEEDUMPs, JAVADUMPs such as threads, stacks, libraries, registers, control blocks, locks, traces, storage, GC, and so forth.

Since SYSTDUMP captures most of the data captured in SVCDUMP and is smaller than SVCDUMP, it has eliminated or reduced the need for SVC dumps captured via SLIP traps. However, taking a console dump is still the most effective way to get a complete snapshot of your Java application, JVM, and the native environment on z/OS, particularly when the Java program is launched in an MVS address space. It provides the information to help you find out the root cause of the problem.

SYSTDUMP and SVCDUMP might contain multiple address spaces. It is important to work with the correct address associated with the failing Java process. “Dump analysis and tools” on page 193 describes the tools that can be used to analyze SYSTDUMPs and SVCDUMPs.

7.3.3 JAVADUMP and HEAPDUMP

JAVADUMP is a file that contains diagnostic information related to the JVM and a Java application captured at a point during execution. The information in this dump can be about the operating system, the application environment, Java threads, stacks, locks, monitors, and memory.

JAVADUMP is enabled in the IBM JVM by default. It is generated when one of the following occurs:

- ▶ A fatal native exception occurs in the JVM (not a Java exception).
- ▶ The JVM has completely run out of heap space. It can be disabled by setting the `IBM_JAVADUMP_OUTOFMEMORY=FALSE` environment variable.
- ▶ A signal is sent to the JVM from the operating system (`kill -3 pid` in USS, `CTRL+V` in z/OS).
- ▶ In the Java code, `com.ibm.jvm.Dump.JavaDump()` is called.

The exact conditions in which you get a JAVADUMP vary depending on the `JAVA_DUMP_OPTS` environment variable. It is possible to get a JAVADUMP when the JVM terminates normally (on an interrupt).

The IBM JVM handles the “fatal” exception by producing a SYSTDUMP followed by a JAVADUMP, and terminates the JVM process. Should anything be wrong during the JAVADUMP creation, you can turn off the JAVADUMP by using the `DISABLE_JAVADUMP` environment variable.

The JAVADUMP is a text file that you can read without any tool. It contains three major distinct sections:

- ▶ Runtime environment
Current thread details, operating environment, application environment, loaded libraries, exception information, and system properties
- ▶ Thread dumps
Full thread dump
- ▶ Monitor information
Monitor pool info, monitor pool dump, JVM system monitor dump, thread IDs, Java object monitor dump

JAVADUMP detail analysis is described in “Dump analysis and tools” on page 193.

HEAPDUMP is an IBM JVM-specific mechanism that generates a dump of all the live objects that are on the Java heap; that is, those that are being used by the running Java application. From IBM SDK for z/OS Java 2 Technology Edition, V1.4 SR2, this dump is stored in a file using phd format. You can use various tools to analyze the dump file to address a problem such as memory leak.

A HEAPDUMP can be generated from a running JVM in the following ways:

- ▶ Explicit generation of a HEAPDUMP
 - By sending a signal (**kill -3 pid** in USS) to the JVM from the operating system
 - By calling the **com.ibm.jvm.Dump.HeapDump()** method inside Java code as long as the IBM_HEAPDUMP environment variable is set
 - By using the JVMRI to request a HEAPDUMP from a loaded agent
- ▶ Triggered generation of a Heapdump
 - A fatal native exception occurs in JVM (not Java exception)
 - An OutOfMemory or heap exhaustion condition occurs

By default, the HEAPDUMP is in binary format. You can get a text format HEAPDUMP by using `opts= stanza` with `-Xdump:heap` or by the existence of an environment variable: `IBM_JAVA_HEAPDUMP_TEST` or `IBM_JAVA_HEAPDUMP_TEXT`.

7.3.4 Other problem determination data

Binary or formatted trace data from the JVM internal high-performance trace provides information about the JVM layer closest to the underlying operating system. This layer maintains an in-storage buffer tracing recent calls. It helps to

identify otherwise hard to spot problems resulting from failing operations such as `pthread_quiesce_and_get_np()` that are used to freeze threads prior to a GC. It can be extracted from SVCDUMP using a tool such as `svcdump.jar`.

Other information such as debugging messages written to `stderr` by JVM or debugging messages from your Java application can also help identify application-specific problems. However, that is not our focus in this chapter.

Some problem determination data can be obtained from a profiling tool such as the profiling tools in IBM Rational Application Developer Version 6 or the JDK profiling tool Hprof. However, they are not as reliable as HEAPDUMP and the performance impact prevents them from being used in a production environment.

7.4 Dump analysis and tools

This section describes the analysis of the following dumps:

- ▶ SYSTDUMP and SVCDUMP
- ▶ JAVADUMP
- ▶ HEAPDUMP

It also discusses tools such as `svcdump.jar` and `HeapAnalyzer`.

A simple Java program is used in this section to generate dumps on z/OS. The program just keeps on creating new objects within an infinite loop until the Java heap is exhausted. The Java heap size is set to 16M to reduce the dump size. The default `JAVA_DUMP_OPTS` is used. Therefore, three dumps are generated (Figure 7-1 on page 194).

```

Session B - [24 x 80]
File Edit View Communication Actions Window Help

ALEX02:/u/alex02: >java -Xmx16m Main
JVMMDG217: Dump Handler is Processing OutOfMemory - Please Wait.
JVMHP002: JVM requesting System Transaction Dump
JVMHP012: System Transaction Dump written to ALEX02.JVM.TDUMP.ALEX025.D051007.T162630
JVMMDG315: JVM Requesting Heap dump file
JVMMDG318: Heap dump file written to /u/alex02/HEAPDUMP.20051007.162636.16777299.phd
JVMMDG303: JVM Requesting Java core file
JVMMDG304: Java core file written to /u/alex02/JAVADUMP.20051007.162638.16777299.txt
JVMMDG274: Dump Handler has Processed OutOfMemory.
JVMST109: Insufficient space in Javaheap to satisfy allocation request
Exception in thread "main" java.lang.OutOfMemoryError: JIT010:OutOfMemoryError,
JIT_ARRAYALLOC failed
    at java.util.ArrayList.ensureCapacity(ArrayList.java(Compiled Code))
    at java.util.ArrayList.add(ArrayList.java(Inlined Compiled Code))
    at Main.addObject(Main.java(Inlined Compiled Code))
    at Main.main(Main.java(Compiled Code))
ALEX02:/u/alex02: >
===> _

INPUT
ESC=¢  1=Help    2=SubCmd  3=HlpRetrn  4=Top      5=Bottom   6=TSO
        7=BackScr  8=Scroll  9=NextSess 10=Refresh 11=FwdRetr 12=Retrieve

MA b 21/007
Connected to remote server/host 9.12.4.126 using lu/pool 5C76T506 ar

```

Figure 7-1 Dumps generation

The three dumps are:

- ▶ MVS data set ALEX02.JVM.TDUMP.ALEX025.D051007.T162630
- ▶ HEAPDUMP file /u/alex02/HEAPDUMP.20051007.162636.1677299.phd
- ▶ JAVADUMP file /u/alex02/JAVADUMP.20051007.162638.1677299.txt

These dumps are transferred to a desktop and analyzed on the desktop in the following dump analysis discussions.

7.4.1 Analyzing SYSTDUMP and SVCDUMP

SYSTDUMP and SVCDUMP are unformatted dumps. They can be analyzed using the same set of tools. The mainframe system programmers are accustomed to the Interactive Problem Control System (IPCS). IBM SDK for z/OS Java 2 Technology Edition, V1.4 comes with a command line tool called jformat. The svcdump.jar from IBM can be used for the same purpose.

For information on how to use IPCS to analyze SYSTDUMP or SVCDUMP, refer to *z/OS MVS Interactive Problem Control System (IPCS) User's Guide*, SA22-7596.

For information on how to use jformat, refer to the IBM JDK Diagnostic Guide (*IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 1.4.2 Diagnostic Guide*, SC34-6358).

The svcdump.jar is a SYSTDUMP and SVCDUMP analysis tool that can be downloaded from the following Web site:

<https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=diagjava>

The svcdump.jar enables direct access to the binary SYSTDUMP and SVCDUMP created on z/OS, without the need for IPCS. The svcdump.jar contains the following parts:

- ▶ *Dump utility* that formats the native and Java stacks for threads in a dumped process that includes an instantiated JVM. It also prints information such as core trace buffer maintained by the JVM and the system trace.
- ▶ *FindRoots utility* provides ways of formatting the object graphs in the Java-managed heap. This is useful for memory leak analysis. It generates a HEAPDUMP phd file which can be further analyzed using some other Heap Analysis tools.
- ▶ *A Java API* that can be used to write ad hoc utilities.

Since the tool is written in Java, it can be run on any platform. If you run it on z/OS, a native library called libsvcdump.so is used to access the dumps in MVS data sets directly, which saves the efforts of copying the dumps to HFS files. If you want to run the tool on your desktop, no native library is needed. But you need to ftp the MVS data set in binary format to your desktop to analyze.

Before using the tool, download the tool and put the svcdump.jar in the current directory. Ftp the dump files and store the files in the current directory as well.

Use the following command to run the dump utility:

```
java -Xmx348m -cp svcdump.jar com.ibm.jvm.ras.svcdump.Dump -options  
<dumpfilename> > > <resultfilename>
```

You can create a shell script with a native library path in USS and run the shell script as either a USS process or an MVS batch job.

Figure 7-2 on page 196 shows a default Dump utility report.

```

defaulttrpt.txt - Notepad
File Edit Format View Help
Dump title: Java J2RE 1.4.2 IBM z/OS Persistent Reusable VM build cm1
Time of dump: Fri Oct 07 12:26:30 EDT 2005
Jvm fullversion: J2RE 1.4.2 IBM z/OS Persistent Reusable VM build cm1

found 13 TCBs in asid 0x42

TCB of unknown type 6fdf30 caa 00000000
TCB of unknown type 6fd358 caa 00000000
TCB of unknown type 6fd0c8 caa 00000000
TCB of unknown type 6ff2a0 caa 00000000

TCB 6ff540 tid 19649838 pthread id 1944835000000000 tid type 0x000000
Dsa      Entry      Offset      Function
---      -
19a42040 7c9015b8 0001e4b0 SYSTDUMP
19a421a0 7c91f538 00000530 sysDump
19a42740 7cb9cba0 000005da dgDumpHandler
19a43600 7cc860f8 0000157e manageAllocFailure
19a436e0 7cc05700 00000324 lockedHeapAlloc
19a437a0 7cc099a8 00000230 allocMiddlewareArray
19a43820 7c5a4e50 0000010e g3_cv_anewarray
19a438c0 1aeef160 000000aa java/util/ArrayList.ensureCapacity(1
19a439c0 1aef5478 000000b2 Main.main([Ljava/lang/String;)V
19a43ae0 7c200c40 ffffffff4 INVEXFER
19a44000 7c5e2ed0 00000058 C_invokerForXfer
19a44080 7cce8d38 000027b0 mmipExecuteJava
19a444a0 7cd05ee0 000005d2 xerunJniMethod
19a44580 7ca903c0 00000134 jni_CallStaticVoidMethod
19a44640 190084f8 00002266 main
19a44720 06ef2120 00000950 CEEVROND
1961d4f0 1911ead8 000000b4 EDCZHINV

```

Figure 7-2 Dump utility default report

The default report of the Dump utility contains:

- ▶ Dump title and time
- ▶ Full version string for the JVM
- ▶ Native and Java stacks for the threads in the process
- ▶ Loaded DLLs
- ▶ Environment variable values
- ▶ LE runtime options values for HEAP and STACK
- ▶ Entries from the systrace for each thread, resolved to a code function in LE dlls

Dump utility behavior can be changed by properties and options controlling the report's generation. The options can include:

- ▶ cache: Print alloc cache

- ▶ `dis <addr><n>`: Disassemble <n> instructions, starting at <addr> (hex)
- ▶ `dump <addr><n>`: Dump <n> words of storage, starting at <addr> (hex)
- ▶ `dumpclasses`: Dump all the classes and their methods
- ▶ `dumpclass <addr>`: Dump the class at <addr>
- ▶ `dumpnative`: Dump all the native methods in all the loaded classes
- ▶ `dumpprops`: Dump all the system properties
- ▶ `exception`: Print old exceptions
- ▶ `heap`: Print a summary of objects that are in the heap
- ▶ `systrace`: Prints the system trace table
- ▶ `r<n>`: Include saved register <n> in the stack trace
- ▶ `tcbsummary`: Print a summary of what the tcbs are doing
- ▶ `verbose`: Print extra information
- ▶ `debug`: Print debug information
- ▶ `version`: Print the Dump tool version

More information about the options is in redpaper *WebSphere for z/OS V5 JVM dump and heap analysis tools*, REDP-3950.

Option `-tcbsummary` collates cases where threads have identical native stacks, as shown in Example 7-1.

Example 7-1 TCB summary report (truncated)

found 13 TCBs in asid 0x42

The following tcbs had this traceback:
6ff540

```
SYSTDUMP
sysDump
dgDumpHandler
manageAllocFailure
lockedHeapAlloc
allocMiddlewareArray
g3_cv_anewarray
java/util/ArrayList.ensureCapacity(I)V
Main.main([Ljava/lang/String;)V
INVFXFER
c_invokerForXfer
mmipExecuteJava
xeRunJniMethod
jni_CallStaticVoidMethod
```

```
main
CEEVROND
EDCZHINV
(unknown)
```

The following tcbs had this traceback:
6e2e88

```
CEEOPCW
CEEVH20S
pthread_cond_wait
ThreadUtils_BlockingSection
sysSignalWait
signalDispatcherThread
xmExecuteThread
threadStart
ThreadUtils_Shell
CEEVROND
CEEOPCMM
(unknown)
CEEOPCMM
(unknown)
```

The following tcbs had this traceback:
6e2cf0 6e2b58

```
CEEOPCW
CEEVH20S
pthread_cond_wait
condWait
sysMonitorWait
lkMonitorWait
JVM_MonitorWait
JVM2JNI
mmipInvokeJniMethod
mmipExecuteJava
xeRunDynamicMethod
threadRT0
xmExecuteThread
threadStart
ThreadUtils_Shell
CEEVROND
CEEOPCMM
(unknown)
CEEOPCMM
(unknown)
```

The thread snapshot is very useful for troubleshooting multi-threading problems. It is also very useful to monitor the threads running in the Java application for the purpose of performance tuning.

Use the following command to run the FindRoots utility:

```
java -Xmx348m -cp svcdump.jar com.ibm.jvm.ras.findroots.<UtilityName>
<filename> > <resultfilename>
```

To extract a HEAPDUMP phd file, you can use the Convert tool of the FindRoots utility. Other tools in the FindRoots utility can be used for further detailed Java Heap analysis.

7.4.2 Interpreting a JAVADUMP

A JAVADUMP file contains tags. This metadata makes it easier to parse and perform simple analysis on the contents of the JAVADUMP file.

Example 7-2 TITLE and XHPI sections (truncated)

NULL	

OSECTION	TITLE subcomponent dump routine
NULL	=====
1TISIGINFO	OUTOFMEMORY received
1TIDATETIME	Date: 2005/10/07 at 16:26:38
1TIFILENAME	Javacore filename:
	/u/alex02/JAVADUMP.20051007.162638.16777299.txt
NULL	

OSECTION	XHPI subcomponent dump routine
NULL	=====
1XHTIME	Fri Oct 7 16:26:38 2005
1XHFULLVERSION	JNI J2RE 1.4.2 IBM z/OS Persistent Reusable VM build
	cm142-20050623
NULL	
1XHOPENV	Operating Environment
NULL	-----
2XHHOSTNAME	Host : WTSC76.ITS0.IBM.COM:9.12.4.126
2XHOSLEVEL	OS Level : z/OS V01 R07.00 Machine 2094 Node SC76
2XHCPUS	Processors -
3XHCPUARCH	Architecture : (not implemented)
3XHNUMCPUS	How Many : (not implemented)
3XHCPUSENABLED	Enabled : 4
NULL	

In Example 7-2, the leftmost 15 columns of each line of the JAVADUMP is the tag. The first digit of the tag is a nesting level. The second and third characters identify the component that wrote the message. The remainder is a unique string.

NULL is a special tag that means the line is purely added for readability. Each section starts with an 0SECTION tag.

The file header (TITLE) shows the general information about the dump. In Example 7-2, an OUTFMEMORY signal has been received.

The XHPI section contains information about:

- Operating system
- User limit
- JVM signal handlers
- Chained signal handlers
- Environment variables
- Thread counts
- Current thread detail

The System Properties (CI) section, shown in Example 7-3, contains information to determine exactly what native executable and Java classes were being run when the dump occurred. The UserArgs shows arguments for JVM, which might be supplied by the user or generated during JVM initialization. The Sys Classpath contains locations from which the Java API is loaded. The classpath is also shown in the report as -Djava.class.path.

Example 7-3 CI Section

```

0SECTION      CI subcomponent dump routine
NULL          =====
1CIJAVAVERSION J2RE 1.4.2 IBM z/OS Persistent Reusable VM build cm142-20050623
1CIRUNNINGAS  Running as a standalone JVM
1CICMDLINE    java -Xmx16m Main
1CIJAVAHOMEDIR Java Home Dir:  /usr/lpp/java/J1.4
1CIJAVADLLDIR  Java DLL Dir:   /usr/lpp/java/J1.4/bin
1CISYSCP      Sys Classpath:
/usr/lpp/java/J1.4/lib/core.jar:/usr/lpp/java/J1.4/lib/graphics.jar:/usr/lpp/java/J1.4/lib/security.jar:/usr/lpp/java/J1.4/lib/server.jar:/usr/lpp/java/J1.4/lib/xml.jar:/usr/lpp/java/J1.4/lib/charsets.jar:/usr/lpp/java/J1.4/classes:/usr/lpp/java/J1.4/lib/ibmcertpathprovider.jar:/usr/lpp/java/J1.4/lib/ibmjcefw.jar:/usr/lpp/java/J1.4/lib/ibmjgssprovider.jar:/usr/lpp/java/J1.4/lib/ibmjssefips.jar:/usr/lpp/java/J1.4/lib/ibmjsseprovider.jar:/usr/lpp/java/J1.4/lib/ibmorbi.jar:/usr/lpp/java/J1.4/lib/ibmorbiapi.jar:/usr/lpp/java/J1.4/lib/ibmpkcs.jar
1CIUSERARGS   UserArgs:
2CIUSERARG    vfprintf 0x19000170
2CIUSERARG    -Xmx16m
2CIUSERARG    -Dinvokedviajava

```

2CIUSERARG	-Djava.class.path=./u/alex02/JRecord.jar:/u/alex02:
2CIUSERARG	vfprintf
NULL	
1CIJVMMI	JVM Monitoring Interface (JVMMI)
NULL	-----
2CIJVMMI0FF	No events are enabled.

Example 7-4 shows the Data Conversion (DC) section. It displays information about internal functions used to convert various character formats and data used to handle Java types.

Example 7-4 DC Section (truncated)

0SECTION	DC subcomponent dump routine
NULL	=====
1DCHEADEREYE	Header eye catcher DCST
1DCHEADERLEN	Header length 24
1DCHEADERVER	Header version 1
1DCHEADERMOD	Header modification 0
1DCINTERFACE	DC Interface at 0x19A6F08C with 15 entries
2DCINTERFACE	1 - dcCString2JavaString 0x19A5E640
2DCINTERFACE	2 - dcInt642CString 0x19A5E670
2DCINTERFACE	3 - dcJavaString2NewCString 0x19A5E6A0
2DCINTERFACE	4 - dcJavaString2CString 0x19A5E6D0
2DCINTERFACE	5 - dcJavaString2NewPlatformString 0x19A5E700
2DCINTERFACE	6 - dcJavaString2UTF 0x19A5E730
2DCINTERFACE	7 - dcPlatformString2JavaString 0x19A5E790
2DCINTERFACE	8 - dcUnicode2UTF 0x19A5E7C0
2DCINTERFACE	9 - dcUnicode2UTFLength 0x19A5E7F0
2DCINTERFACE	10 - dcUTF2JavaString 0x19A5E820
2DCINTERFACE	11 - dcUTFClassName2JavaString 0x19A5E850
2DCINTERFACE	12 - dcJavaString2ClassName 0x19A5E760
2DCINTERFACE	13 - dcUTF2UnicodeNext 0x19A5E880
2DCINTERFACE	14 - dcVerifyUTF8 0x19A5E8B0
2DCINTERFACE	15 - dcDumpRoutine 0x19564E78
1DCARRAYINFO	Array info at 0x19A5EBC0 with 16 entries
2DCARRAYINFO	1 - index 0 signature 0 name N/A factor 0
2DCARRAYINFO	2 - index 0 signature 0 name N/A factor 0
2DCARRAYINFO	3 - index 2 signature L name class[] factor 4
2DCARRAYINFO	4 - index 0 signature 0 name N/A factor 0
2DCARRAYINFO	5 - index 4 signature Z name bool[] factor 1
2DCARRAYINFO	6 - index 5 signature C name char[] factor 2
2DCARRAYINFO	7 - index 6 signature F name float[] factor 4

Example 7-5 shows the Diagnostics Settings (DG) section. It contains the information about the size of the buffer used to hold the JAVADUMP before being flushed to disk and the information about the JVM trace settings.

Example 7-5 DG Section

OSECTION	DG subcomponent dump routine
NULL	=====
1DGTRCENABLED	Trace enabled: Yes
2DGTRCTYPE	Trace: Internal
2DGTRCBUFFERS	Buffer specification: 8k
2DGTRCBUFALLOC	Buffers allocated: 0
2DGTRCBUFUSED	Buffers in use: 0
1DGJDUMPBUFF	Javdump buffer size (allocated): 2621440

The Storage Management (ST) section (Example 7-6) gives various storage management values, including:

- If concurrent mark works
- Current heap address limits
- Counts of allocation failure (AF counter) and Garbage Collection cycles (GC Counter)
- Free space in heap and size of the current heap

Example 7-6 ST Section

OSECTION	ST subcomponent dump routine
NULL	=====
1STGCMODES	Resettable GC: No
1STGCMODES	Concurrent GC: No
1STCURHBASE	Current Heap Base: 0x19B301FC
1STCURHLIM	Current Heap Limit: 0x1AB1FBFC
1STMWHBASE	Middleware Heap Base: 0x19B301FC
1STMWHLIM	Middleware Heap Limit: 0x1AB1FBFC
1STGCHELPERS	Number of GC Helper Threads: 3
1STJVMOPTS	-Xconcurrentlevel: 0
1STJVMOPTS	-Xconcurrentbackground: 0
1STGCCTR	GC Counter: 9
1STAFCTR	AF Counter: 9
1STHEAPFREE	Bytes of Heap Space Free: 26eda0
1STHEAPALLOC	Bytes of Heap Space Allocated: fefa00
1STSMBASE	SM Base: 0x0
1STSMEND	SM End: 0x0
1STPAMSTART	PAM Start: 0x0
1STPAMEND	PAM End: 0x0
1STCOMACTION	Compact Action: 1

The Execution Engine (XE) section (Example 7-7) provides information such as JIT initialization and the JIT mixed-mode compilation threshold (800).

Example 7-7 XE Section

OSECTION	XE subcomponent dump routine
NULL	=====
1XETHRESHOLD	MMI threshold for java methods is set to 800
1XEJITINIT	JIT is initialized
1XEJVMPIOFF	JVMPI is not activated
1XEJNITHRESH	MMI threshold for JNI methods is set to 0
1XETRCHIS	Trace history length is set to 4
1XEJITDUMP	JIT dump routine is not yet implemented.

Example 7-8 shows the locks, monitors and deadlocks (LK) section. A lock prevents more than one entity from accessing a shared resource. In a JVM, threads compete for various resources in the JVM and locks on Java objects. A monitor is a special kind of locking mechanism that is used in the JVM to allow flexible synchronization between threads.

A flat monitor is a guardian flag to indicate that a piece of code is locked. However, if another thread wants to access code that is locked, the JVM has to create a monitor object to hold the second thread and arrange for a signaling mechanism to coordinate access to the code section. This monitor is called *inflated monitor*.

The LK section is split into:

- Monitor pool info
- Monitor Pool Dump
- JVM system monitor dump
- Thread identifiers
- Java object monitor dump

The LK component dump can be used to diagnose a thread deadlock. JAVADUMP automatically diagnose most thread deadlocks.

Example 7-8 LK Section

OSECTION	LK subcomponent dump routine
NULL	=====
NULL	
1LKPOOLINFO	Monitor pool info:
2LKPOOLINIT	Initial monitor count: 32
2LKPOOLEXPNUM	Minimum number of free monitors before expansion: 5
2LKPOOLEXPBY	Pool will next be expanded by: 16

```

2LKPOOLTOTAL      Current total number of monitors: 32
2LKPOOLFREE       Current number of free monitors: 30
NULL
1LKMONPOOLDUMP    Monitor Pool Dump (flat & inflated object-monitors):
2LKMONINUSE       sys_mon_t:0x19653938 infl_mon_t: 0x196533B0:
3LKMONOBJECT      java.lang.ref.Reference$Lock@19BB0610/19BB0618: <unowned>
3LKNOTIFYQ        Waiting to be notified:
3LKWAITNOTIFY     "Reference Handler" (0x197ED5A0)
2LKMONINUSE       sys_mon_t:0x19653990 infl_mon_t: 0x196533DC:
3LKMONOBJECT      java.lang.ref.ReferenceQueue$Lock@19BB0348/19BB0350:
<unowned>
3LKNOTIFYQ        Waiting to be notified:
3LKWAITNOTIFY     "Finalizer" (0x197EF090)
NULL
1LKREGMONDUMP     JVM System Monitor Dump (registered monitors):
2LKREGMON         JITC CHA lock (0x198FCCF8): <unowned>
2LKREGMON         JITC Register_Committed_Code lock (0x198FB808): <unowned>
2LKREGMON         Free Class Loader Cache Entry lock (0x19655990): <unowned>
2LKREGMON         IO lock (0x19655938): <unowned>
2LKREGMON         Evacuation Region lock (0x19655888): <unowned>
2LKREGMON         Heap Promotion lock (0x19655830): <unowned>

```

The threads and stack trace (XM) section in Example 7-9 has a complete list of Java threads that are alive. A Java thread is implemented by a native thread of the operating system. The properties of a thread are name, identifier, JVM data structure address, current state, native thread identifier and priority. The values of state can be:

- ▶ R - Runnable
The thread is able to run when given the chance
- ▶ CW - Condition Wait
The thread is waiting
- ▶ MW - Monitor Wait
- ▶ S - Suspended thread
- ▶ MS - Monitor Suspended

Combined with the LK section, the XM section can be used to detect threading issues and find the root cause of a failure.

Example 7-9 XM section (truncated)

```

OSECTION          XM subcomponent dump routine
NULL              =====
NULL
1XMCURTHDINFO     Current Thread Details
NULL              -----

```

3XMTTHREADINFO	"main" (TID:0x19B9B9B8, sys_thread_t:0x19649838, state:R,
4XESTACKTRACE	at java.util.ArrayList.ensureCapacity
4XESTACKTRACE	at java.util.ArrayList.add(ArrayList.java
4XESTACKTRACE	at Main.addObject(Main.java(Compiled Code))
4XESTACKTRACE	at Main.main(Main.java(Compiled Code))
3XHNATIVESTATE	Native Thread State: USER RUNNING
3XHNATIVEDATA	Native Stack Data: base: 19a44e04 top: 0 pointer: 19ac4e04
3XHMONTORHELD	Monitors Held: (not implemented)
NULL	
3XHNATIVESTACK	Native Stack
NULL	-----
3XHTRACEBACK	Program Name Entry Name
3XHTRACEBACK	/u/sovbld/142/builds/cm142/cm142-20050623/src/hpi/

The classloaders and classes (CL) section in Example 7-10 includes classloader summaries and classloader loaded classes. It can be helpful to address classloading related problems.

Example 7-10 CL Section

OSECTION	CL subcomponent dump routine
NULL	=====
1CLCLASSPATH	Classpath
	Z(/usr/lpp/java/J1.4/lib/core.jar),Z(/usr/lpp/java/J1.4/lib/graphics.jar),Z(/usr/lpp/java/J1.4/lib/security.jar),Z(/usr/lpp/java/J1.4/lib/server.jar),Z(/usr/lpp/java/J1.4/lib/xml.jar),Z(/usr/lpp/java/J1.4/lib/charsets.jar),Z(/usr/lpp/java/J1.4/lib/ibmcertpathprovider.jar),Z(/usr/lpp/java/J1.4/lib/ibmjcefw.jar),Z(/usr/lpp/java/J1.4/lib/ibmjgssprovider.jar),Z(/usr/lpp/java/J1.4/lib/ibmjssefips.jar),Z(/usr/lpp/java/J1.4/lib/ibmjsseprovider.jar),Z(/usr/lpp/java/J1.4/lib/ibmorb.jar),Z(/usr/lpp/java/J1.4/lib/ibmorbapi.jar),Z(/usr/lpp/java/J1.4/lib/ibmpkcs.jar)
1CLFLAGOLDJAVA	Oldjava mode false
1CLFLAGBOOTSTRP	Bootstrapping false
1CLFLAGVERBOSE	Verbose class dependencies false
1CLENUMVERIFY	Class verification VERIFY_REMOTE
1CLPNTRNAMECLO	Namespace to classloader 0x00000000
1CLPNTRCHAINLO	Start of cache entry pool 0x198ED250
1CLPNTRCHFREE	Start of free cache entries 0x198EE530
1CLPNTRMETHOTB	Location of method table 0x19755020
1CLPNTRANCHRGLN	Global namespace anchor 0x1964ADF4
1CLPNTRCLSLOADS	System classloader shadow 0x196F9058
1CLPNTRSYSLOADS	Classloader shadows 0x198E7758
1CLPNTRCLSEXT	Extension loader 0x19B9B728

Example 7-11 shows the End section of the JAVADUMP.

Example 7-11 End section

OSECTION	Javdump End section
1DGJDUMP	Javdump Buffer Usage Information
NULL	=====
2DGJDUMPALLOC	Javdump buffer size (allocated): 2621440
2DGJDUMPUSED	Javdump buffer size (used) : 51796

More information about the JAVADUMP can be found in the IBM JDK Diagnostic Guide (*IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 1.4.2 Diagnostic Guide*, SC34-6358).

7.4.3 HeapAnalyzer

IBM HeapAnalyzer is provided as an “as-is” tool to analyze IBM Java heap dumps in the IBM JDK 1.3.1 and higher. It does the following:

- ▶ Creates a tree from HEAPDUMP
- ▶ Calculates the size of each object and the total size of each subtree
- ▶ Finds size drop in a subtree
- ▶ Shows Gap by size and Gap distribution
- ▶ Shows objects by size, total size, number of child, detailed information
- ▶ Shows type by size, count, in alphabetical order
- ▶ Finds type
- ▶ Bookmarks tree navigation
- ▶ Saves and loads processed heap dumps
- ▶ Locates possible leak suspects

You can download the HeapAnalyzer from:

<https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=AW-0IN>

Once you download the ha135.zip file, just unzip it into any folder on your desktop. Make sure you have JRE 1.4.1 or higher to run this tool.

Usage <Java path>java -Xmx[heapsize] -jar ha<HeapAnalyzer version>.jar

For example:

java -Xmx1000m -jar ha135.jar

If there is a java.lang.OutOfMemoryError while you are processing heap dumps, try increasing the maximum heap size (-Xmx) value to give the JVM more memory.

The maximum heap size should not be larger than the size of the available physical memory for this tool due to performance considerations.

The input to the HeapAnalyzer is the phd file, either generated from the JVM or extracted from SYSTDUMP/SVCDUMP.

Select **File** → **Open**, then select the phd file you want to open from the FileOpen dialog. HEAPDUMP.20051007.160406.67108938.phd is opened as an example. The phd file open and analyze process loads the HEAPDUMP, parses each object and class information, and creates graphs based on parsed information. The results are as shown in Figure 7-3.

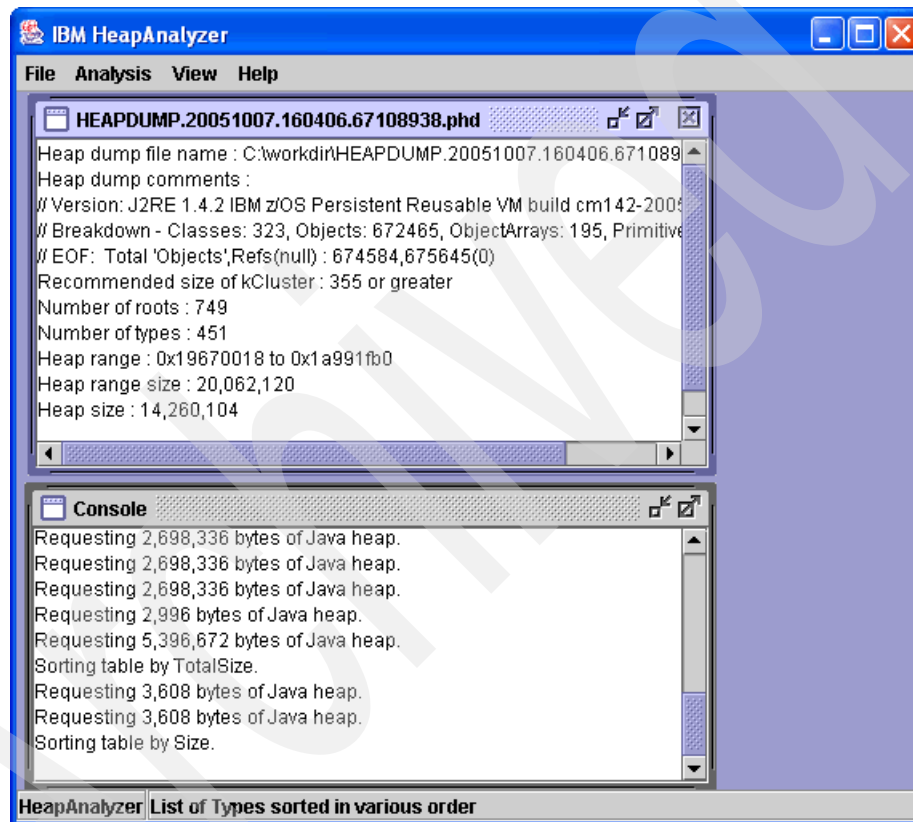


Figure 7-3 Open a phd file result

Do not close the phd file processing status window until you no longer need this HEAPDUMP.

Select **Analysis** → **Tree view**. You should see the tree view as shown in Figure 7-4 on page 208. It shows all the objects in the HEAPDUMP as a tree.

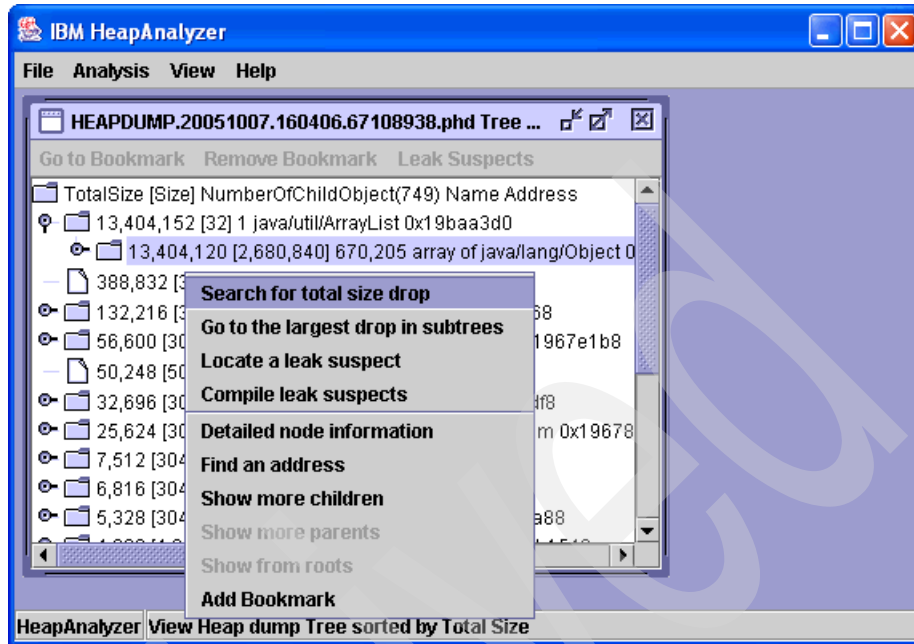


Figure 7-4 Object tree view

Each tree node is in the format:

TotalSize [Size] NumberOfChildObject Name Address

The object tree virtualizes the relationship between all the Java objects in the heap.

Through the object tree context menu, you can search for total size drop and largest drop in subtrees. These features can be used to help analyze memory leaks.

More directly, click **Locate a leak suspect** in the context menu. The tool will analyze the object tree and point out memory leak suspects in the tree. Figure 7-5 on page 209 shows how to locate a memory leak suspect. A memory leak list can be compiled from the memory leak suspects (Figure 7-6 on page 209).

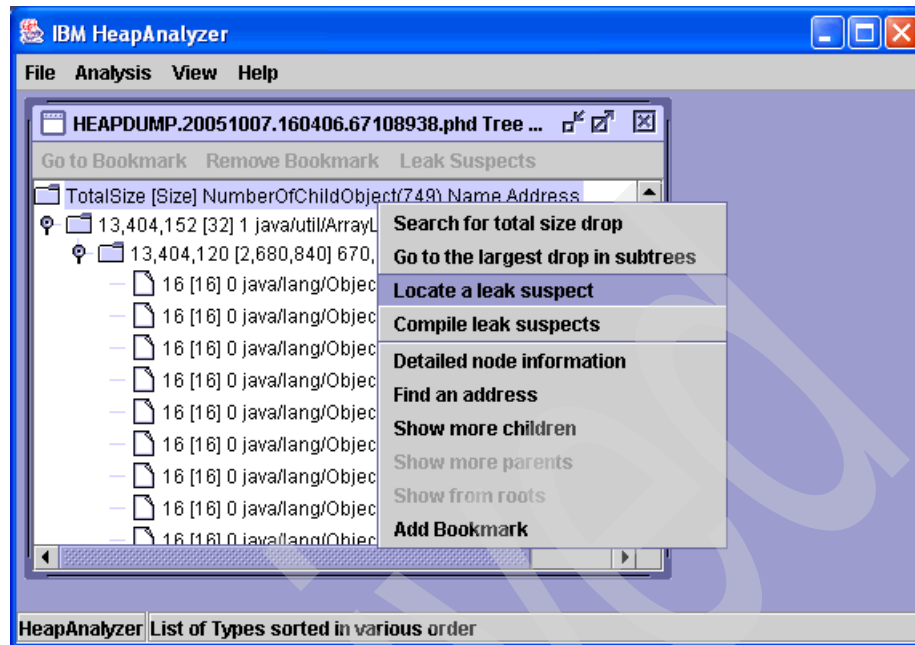


Figure 7-5 Locate a leak suspect

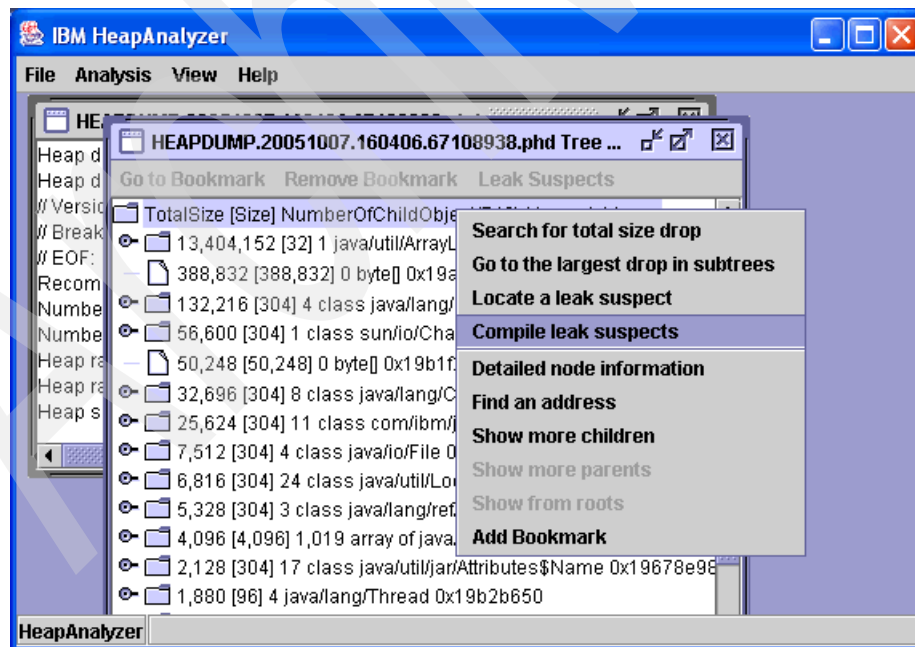


Figure 7-6 Compile leak suspects

The compiled leak suspects list provides a list of object tree node links that allows you to directly jump to a leak suspect to analyze the node in detail. Figure 7-7 shows a compiled leak list. The list has one item that is an array of objects with a size of 13,404,120 bytes.

The basic idea about Java heap analysis using the object tree view is to look for areas where there is an excessive number of children with a large difference in totalsize between parent and child. That is where the memory leaks are most likely located.

Of course you can also get the memory utilization information for different objects in your application from the object tree view. Based on the information, you can try to improve the resource utilization of the application, which, in turn, improves the QoS of your application.

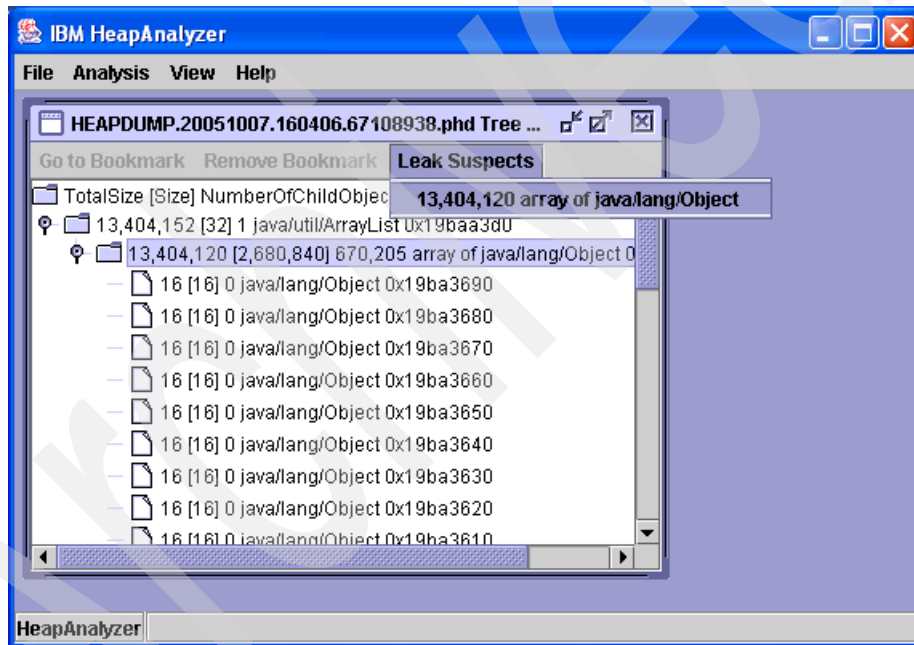


Figure 7-7 Compiled leak suspects

Should you need detailed node information about an object tree node, click **Detailed node information** and details about the object tree node are displayed (Figure 7-8).

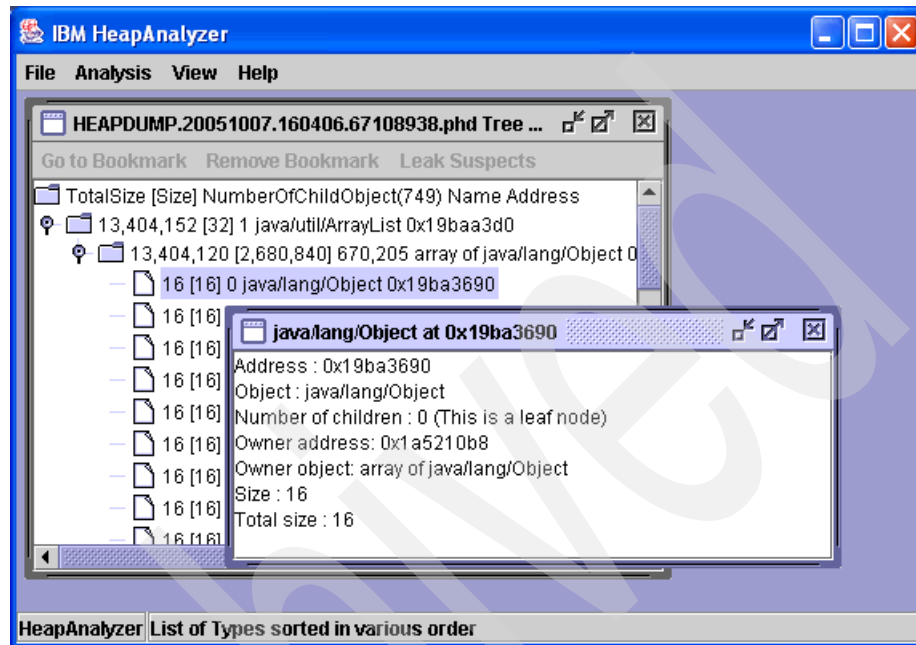


Figure 7-8 Detailed node information

Select **Analysis** → **Objects List** to get the objects list shown in Figure 7-9. It is a different view of all the objects in the Java heap. The list can be sorted by the columns defined in the view.

TotalSize	Size	No.Child	Address	Object
13,404,120	2,680,840	670,205	0x1a5210b8	array of jav...
4,096	4,096	1,019	0x19bb1540	array of jav...
56,208	2,064	281	0x19b442c8	array of jav...
6,952	424	102	0x19b2c530	array of [C
12,024	400	40	0x19b43258	array of jav...
6,816	304	24	0x1967ed98	class java/...
2,128	304	17	0x19678e98	class java/...
11,800	136	15	0x19b2b728	sun/misc/L...
4,464	128	15	0x19b2b6a8	sun/misc/L...
6,120	144	14	0x19b73d90	array of jav...
80	80	12	0x19b97ec8	array of jav...
3,256	64	12	0x19b73a58	array of jav...
80	80	12	0x19b87ce0	array of jav...
25,624	304	11	0x19678d68	class com/i...
2,832	144	10	0x19b36b10	array of jav...
616	304	10	0x1967de28	class sun/i...
6,944	80	9	0x19b87d30	array of jav...

Figure 7-9 Object list

Click **Analysis** → **Types List**, to get the list shown in Figure 7-10. It presents the objects in the Java heap by object type. The list can be sorted by the columns in the view.

Sum of sizes	Count	Type
10,723,392	670,212	java/lang/Object
2,686,112	22	array of java/lang/Ob...
530,928	16	byte[]
125,728	1,578	char[]
43,744	1,367	java/lang/String
14,784	462	java/util/HashMap\$E...
8,256	2	short[]
3,368	4	int[]
3,168	14	array of java/util/Has...
2,448	102	array of [C
2,400	75	java/util/Hashtable\$...
1,792	28	java/net/URL
1,400	17	array of java/util/Has...
880	22	java/util/Locale
800	20	java/lang/ref/Finalizer
784	14	java/util/HashMap
768	16	java/util/Hashtable

Figure 7-10 Types list view

To analyze the Java heap using either the Objects List or the Types List, sort the lists by count. Pay attention to excessive numbers of objects; they might be the memory problem candidates. You can get the same result as in the leak suspects.

The analysis results can be used to improve the application's memory utilization efficiency.

The *Gap Analysis* view provides an object distribution map in the Java Heap, which helps in troubleshooting GC problems.

HeapAnalyzer has several Find and Search functions to help the Java heap analysis.

You cannot diagnose all problems by analyzing HEAPDUMP. GC trace is another source of information to figure out what is going on with Java heap and GC.

7.4.4 Other analysis tools

There are many other dump analysis tools, such as jextract, Dumpviewer, GlowCode, HAT, HeapWizard, JProbe, JSwat, Process Explorer, and so forth. More information about these tools is in *IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 1.4.2 Diagnostic Guide*, SC34-6358.

7.5 Problem determination scenarios

This section gives an overview of problem determination scenarios.

7.5.1 Diagnose crashes

A *crash* should occur only because of a fault in the JVM, or because of a fault in JNI code that is being run inside the Java process. SYSTDUMP and JAVADUMP are collected to determine the failing functions and find the root cause. The svcdump.jar is used for dump analysis.

7.5.2 Debugging hangs

A *hang* refers to a process that is still present, but that has become unresponsive. The hang can be caused by:

- ▶ The process has been deadlocked. In this situation, usually the process does not take up CPU time.
- ▶ The process has become caught in an infinite loop. Usually, the process takes up high CPU time.
- ▶ The process is suffering from bad performance. This is not a real hang.

The deadlock can be found from the LK and XM sections of JAVADUMP. If no deadlock exists but the process consumes CPU time, take a console dump. This can be analyzed using tools in svcdump.jar to determine the looping thread.

If you have no evidence of a deadlock or an infinite loop, the lack of response might be caused by putting threads into explicit sleep, by excessive lock contention, long GC, and so forth.

7.5.3 Debugging memory leaks

A *memory leak* can be either a native (C/C++) memory leak or a Java object leak. LE HEAP usage must be analyzed for a native memory leak. HEAPDUMP and verbose GC are analyzed to pinpoint Java object leak problems.

7.5.4 Debugging performance problems

Check whether the JIT compiler is activated. Check whether the system is tuned to cope with the Java managed heap size specified to avoid excessive paging. Check whether the GC pause time is too long. The verbose GC can be turned on for debugging.

Application design for performance and application profiling is discussed elsewhere in this book.

Archived

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/SG247177>

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, SG247177.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

<i>File name</i>	<i>Description</i>
JRecord Plugin.zip	A zip file with the record Bean generator plugin for Eclipse.
JRecordRuntime.jar	A jar file with required runtime classes for the record Bean generator plugin.

examples.zip

All Java sample code used throughout the book.

System requirements for downloading the Web material

Please follow the instructions and prerequisites explained in the book to use the sample code.

Note: The content of these files is provided in EBCDIC. If you want to use the files, you should upload them to a z/OS system and unpack them there.

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 220. Note that some of the documents referenced here may be available in softcopy only.

- ▶ *ABCs of z/OS System Programming Volume 1*, SG24-6981
- ▶ *zSeries Application Assist Processor (zAAP) Implementation*, SG24-6386
- ▶ *VSAM Demystified*, SG24-6105
- ▶ *WebSphere for z/OS V5 JVM dump and heap analysis tools*, REDP-3950

Other publications

These publications are also relevant as further information sources:

- ▶ *z/OS Language Environment Debugging Guide*, GA22-7560
- ▶ *z/OS MVS Interactive Problem Control System (IPCS) User's Guide*, SA22-7596
- ▶ *IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 1.4.2 Diagnostic Guide SC34-6358* (often referred to as the *IBM JDK Diagnostic Guide*)
- ▶ S. Comella-Dorba, K. Wallnau, R. Seacord and J. Robert, *A Survey of Legacy System Modernization Approaches*, Carnegie Mellon Software Engineering Institute, Technical Note CMU/SEI-2000-TN-003

<http://www.sei.cmu.edu/publications/documents/00.reports/00tn003.html>

Online resources

These Web sites and URLs are also relevant as further information sources:

- ▶ IBM SDK for z/OS, Java 2 Technology Edition, V1.4
<http://www-03.ibm.com/servers/eserver/zseries/software/java/>
- ▶ Java Record I/O (JRIO) Overview
<http://www-03.ibm.com/servers/eserver/zseries/software/java/jrio/overview.html>
- ▶ z/OS Performance: Capacity Planning Considerations for zAAP Processors
<http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP100417>
- ▶ Eclipse
<http://www.eclipse.org/>
- ▶ Eclipse Test & Performance Tools Platform Project (TPTP)
<http://www.eclipse.org/tptp/index.html>
- ▶ Rational Application Developer
<http://www-306.ibm.com/software/awdtools/developer/application/>
- ▶ IBM System z9 and eServer zSeries Application Assist Processor (zAAP)
<http://www-03.ibm.com/servers/eserver/zseries/zaap/>
- ▶ IBM eServer zSeries
<http://www-03.ibm.com/servers/eserver/zseries/>
- ▶ HeapAnalyzer
<http://www.alphaworks.ibm.com/tech/heapanalyzer>
- ▶ HeapRoots
<http://www.alphaworks.ibm.com/tech/heaproots>
- ▶ Diagnostic Guide
<http://www-128.ibm.com/developerworks/java/jdk/diagnosis/>

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

Archived

Index

Symbols

_CEE_DMPTARG 189

A

accurate analysis 112

Ant

- Buildfile 63
- classpath 63
- configuring options 64
- editor 63
- example 66
- jakarta-oro-2.0.8.zip 65
- properties file, example 69
- set up in Eclipse 65
- target 63
- tasks 63
- view 65

Apache Ant

- information 62

Application Assist Processors (zAAP) ix

B

BPXBATCH 18

- address spaces involved 187
- DD cards 18
- environment variables, setting 19
- example 18
- limitations 19–20
- PGM command 18
- return code 28
- SH command 18

BPXBATSL 21

- example 21
- return code 28

Byte Code Instrumentation (BCI) 155

C

CEEDUMP 188–190

COND parameter 13

container 2

converting bytes to Java types 50

D

Data Definition (DD) 13

DD statements 13

DISABLE_JAVADUMP environment variable 191

DLL, failure to load 186

DOM 6

DSA 190

dump analysis tools 214

dump utility 195

 default report 196

 properties and options 196

dump, events triggering 188

E

Eclipse

 downloading 113

 information 62

Eclipse Modeling Framework (EMF) 115

elimination analysis 95

exception 186

EXEC 13

Extensible Markup Language (XML) 6

External Throughput Rate (ETR) 111

F

Financial Information Exchange protocol (FIX) 6

Financial product Markup Language (FpML) 6

FindRoots utility 195

FindRoots utility, finding 199

fractional analysis 94, 96

G

garbage collection 186

garbage collection policy 94

H

Heap size 186

 maximum 186

 maximum default 186

HeapAnalyzer 206

 ha135.zip file 206

HEAPDUMP

- format 192
- ways of generating 192
- Hyper Text Transfer Protocol (HTTP) 6

I

- IBM eServer System z Application Assist Processor (zAAP) 174
- IBM_JAVA_HEAPDUMP_TEST 192
- IBM_JAVA_HEAPDUMP_TEXT 192
- IDCAMS 41
- inflated monitor 203
- Inter Process Communicaiton (IPC) 126
- Interactive Problem Control System (IPCS) 190
- Internal Throughput Rate (ITR) 111
- IPCSHMMPAGES 130
- IPCSHMSPAGES 130
- IRXJCL 14
- ISO 15022XML 6

J

- Java Record I/O (JRIO) 32
 - Directory class 32
 - example, accessing records randomly 38
 - example, copying a data set 35
 - IConstants interface 34
 - IDirectory interface 32
 - IFileInputRecordStream 33
 - IFileOutputRecordStream 33
 - IKeyedAccessRecordFile interface 42
 - IRandomAccessRecord 34
 - IRecordFile interface 32
 - KeyedAccessRecordFile class 42
 - packed decimal numbers, reading 46
 - RecordFile class 32
 - supported file systems 32
 - tracing, enabling 41
 - VSAM types supported 41
- Java Virtual Machine Profiling Interface (JVMPi) 128
- Java, advantages 5
- JAVA_DUMP_HDUMP_PATTERN 189
- JAVA_DUMP_OPTS 188–190
- JAVA_DUMP_OPTS environment variable 191
- JAVA_DUMP_TDUMP_PATTERN 189
- JAVA_DUMP_TOOL 188–189
- JAVADUMP 191
 - classloaders and classes (CL) section 205
 - conditions for 191

- Data Conversion (DC) section 201
- End section 206
- Execution Engine (XE) section 203
- file header (TITLE) 200
- format 200
- locks, monitors and deadlocks (LK) section 203
- sections 192
- Storage Management (ST) section 202
- System Properties (CI) section 200
- tags 199
- threads and stack trace (XM) section 204

JCL

- ABEND 26
- COND parameter 25
- IF-ELSE 26
- MEMLIMIT parameter 25
- RC 26
- REGION parameter 25
- RUN 26
- TIME parameter 25

- JES2 14

- JES3 14

- JIT compiler 186

- job card 13

- Job Control Language (JCL) 13

- Job Entry Subsystem (JES) 14

JVM

- dump environment variables 188
- dump types 188
- private storage 186

L

- LE HEAP 186

- LEDATA 190

- libsvcdump.so 195

- Local debugging 89

M

- manually installing TPTP 115

- MAXSHAREPAGES 130

- multi-threading problems 199

O

- OutOfMemoryError 186

P

- packed decimal numbers 46

- converting from Java 49
 - converting to Java 46
- performance analysis, techniques 94
- performance goals, for batch applications 93
- performance tools, categories 92
- phd format 192
- problems
 - areas to be checked 184
 - collecting data 187
- profiling
 - filtering 127
 - remote 126
 - standalone 126
- profiling in Eclipse
 - code coverage 134
 - connection with host, creating 131
 - connection with host, testing 131
 - creating a new configuration 131
 - memory analysis 134
 - starting 130
 - time analysis 135
- profiling, launch a Java program with profiling enabled 149
- profiling, LIBPATH 149

R

- Rational Application Developer (RAD)
 - debugging 90
- Redbooks Web site 220
 - Contact us xi
- REGION 13
- Remote Agent Controller
 - configuration file 129
 - launching a program 149
 - LIBPATH 149
 - listener ports, changing 129
 - listener ports, checking 129
 - shared memory configuration 130
 - shared memory, BPXPRMxx parameters 130
 - starting 129
- Remote Agent Controller for z/OS
 - architectural overview 123
 - dependency 124
- remote debugger 82
 - disconnecting 87
- remote debugging
 - basic steps 72
 - Classfile generation options 75

- JVM options 76
 - preferences, setting up 74
 - shell script, example 76
 - shell script, with debugging options 75
- return code, passing from a Java program 28

S

- SAX 6
- screen scraping 4
- service-oriented architecture (SOA) 6
- simple object access protocol (SOAP) 6
- Standards for Technology in Automotive Retail (STAR) 6
- statistical analysis 94–95
- stubbing 95
- svcdump.jar 195
- svcdump.jar, parts 195
- Symbolic link, in USS 186
- SYSTDUMP 190
- System Display and Search Facility (SDSF) 15
 - Processes Panel (PS) 17
 - Status Panel (ST) 16
- System Display and Search Facility (SDSF), useful commands 16
- System.currentTimeMillis() 97

T

- tcbsummary 197
- TERMTHDACT 190
- Test and Performance Tools Platform (TPTP) 112
- The 111
- This 95
- thread, properties 204
- Timer class, reporting example 100
- Timer.getTimer() 99
- TIMEUSED macro 103
- TPTP
 - automated installation 116
 - install 112, 116
- Transactions Per Second (TPS) 93
- TRAP 190
- TSO login size 186

U

- Universal Description, Discovery and Integration (UDDI) 6
- USERABEND 188

Using 96

V

VSAM

- characteristics of data set, specifying 45
- cluster 41
- key 41
- logical record 41
- methods 43
- reading the current record 43
- readSequentially(...) method 43
- setting a reference to a key 43
- types 41

- HFS file access with JNI 179
- type conversions using JRecord 179
- VSAM access with JRIO 180
- utilization estimates, for typical operations 178
- utilization potentials 175
- zAAP projection tool
 - output, example 177
 - spreadsheet 177

W

Web Service Description Language (WSDL) 6

WebSphere Application Server (WAS) 7

WebSphere Extended Deployment (XD) 7

WLM

- Service Class 186

X

-Xdebug 76

-Xdump 188

-Xifa 174

XML Schema Infoset Model (XSD) 115

-Xrunjdw 76

Z

z/OS Language Environment (LE) Runtime Options (RTOs) 185

z/OS Work Load Manager (zWLM) 186

zAAP

- introduction 174

- JVM runtime options 174

- starting the projection tool 177

- utilization

- CTG/CICS access 176

- IMS Connect 176

- Trade 2 176

- Trade 3 176

- traditional IMS 176

- utilization estimates

- DB2 access 181

- HFS file access with Java I/O library 179

- HFS file access with Java I/O library and JNI 180

Java Stand-alone Applications on z/OS Volume 1

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



Redbooks

Java Stand-alone Applications on z/OS Volume 1

Setting up an environment for managing Java programs on z/OS

Building, deploying, running, and debugging applications

Analyzing application performance and exploiting zAAP

This IBM Redbook is about using Java on z/OS to develop, deploy, and run stand-alone and batch applications.

Java has certainly become the most popular programming language, and it is the language of choice for many enterprise applications today. This is the result of many advancements in the language itself as well as related technologies and tools that enable users to develop and run cost-effective, scalable, high-performance solutions.

IBM System z servers provide comprehensive support for deploying Java applications while providing the traditional quality of service, advanced security, and scalability that users have come to expect from this server family. IBM System zApplication Assist Processors (zAAP) further enhance the ability of an enterprise to implement complex and heavy workload applications on System z in a cost-effective and reliable manner.

There are many technical and business-oriented documents that describe the value of running J2EE applications on System z using WebSphere Application Server. However, less attention has been given to running stand-alone Java applications. This book shows how Java can leverage the power of IBM System z to run stand-alone and batch applications.

This book is a technical guideline for Java application developers who do not have prior experience with the System z operating environment, as well as System z operators who are new to the world of Java because of the inevitable transition occurring in the IT industry. Chapters in this book are designed to be self-contained so that they can be read in any order.

**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