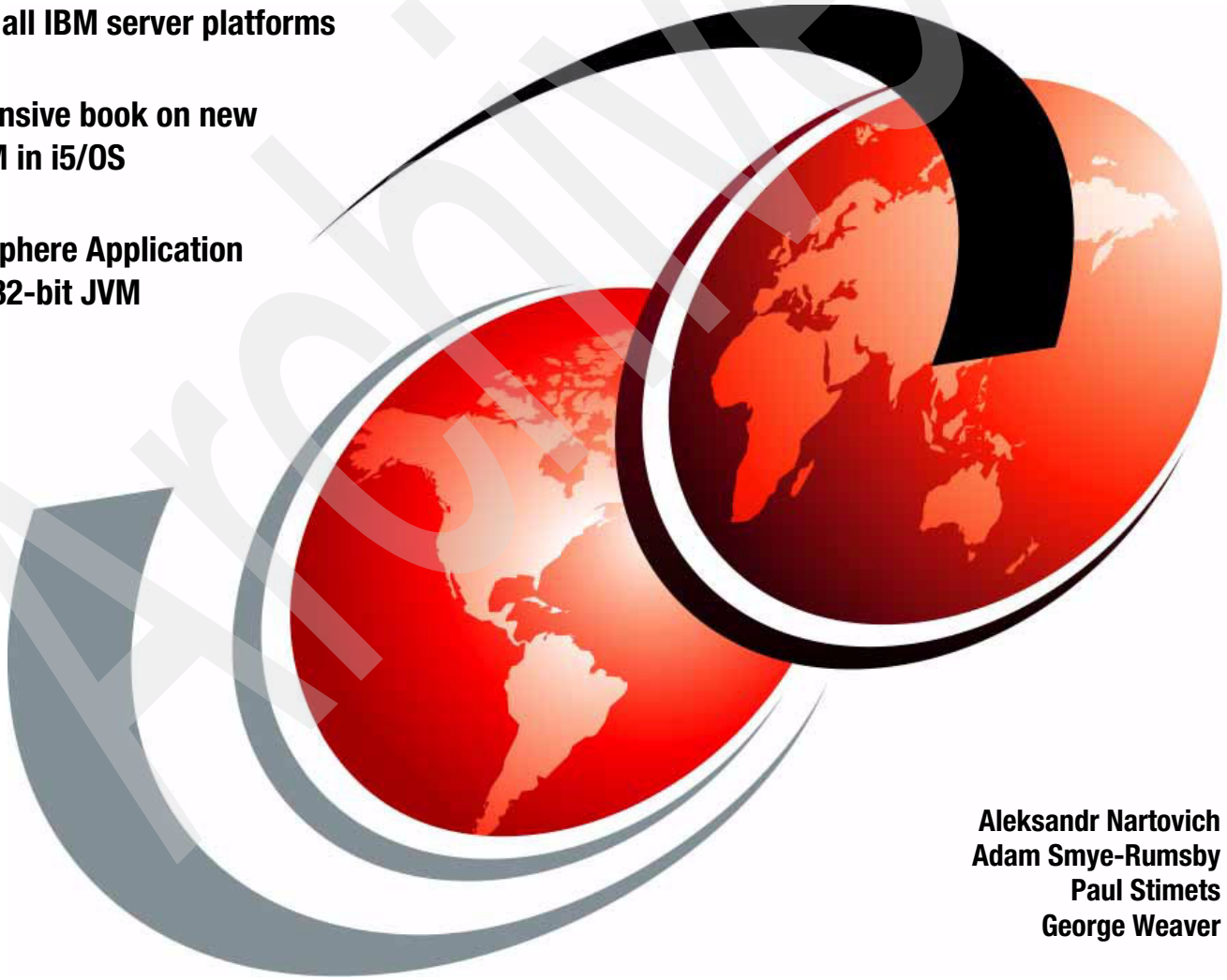
IBM

IBM Technology for Java Virtual Machine in IBM i5/OS

- Most information about 32-bit JVM applies to all IBM server platforms
- Comprehensive book on new 32-bit JVM in i5/OS
- Run WebSphere Application Server in 32-bit JVM



Aleksandr Nartovich
Adam Smye-Rumsby
Paul Stimets
George Weaver

Redbooks

ibm.com/redbooks

International Technical Support Organization

IBM Technology for Java Virtual Machine in IBM i5/OS

February 2007

SG24-7353-00

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

First Edition (February 2007)

This edition applies to IBM® Developer Kit and Runtime Environment, Java™ 2 Technology Edition, Version 5.0.

Contents

Notices

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

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

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

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

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

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

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

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

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

COPYRIGHT LICENSE:

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

Trademarks

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

Redbooks (logo) ™
alphaWorks®
developerWorks®
eServer™
iSeries™
i5/OS®
z/OS®
z9™
AIX®
AS/400®
Domino®
DB2 Universal Database™
DB2®
Integrated Language Environment®
IBM®
Language Environment®
Lotus®
OS/390®
OS/400®
PowerPC®
POWER™
Rational®
Redbooks™
System i™
System i5™
System p™
System p5™
System z™
System z9™
Tivoli®
WebSphere®

The following terms are trademarks of other companies:

Enterprise JavaBeans, EJB, Java, Javadoc, JavaBeans, JavaServer, JavaServer Pages, JDBC, JDK, JMX, JRE, JSP, JVM, J2EE, J2ME, J2SE, Solaris, Sun, Sun Microsystems, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

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

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 gives a broad understanding of a new 32-bit Java™ Virtual Machine (JVM™) in IBM i5/OS®. With the arrival of this new JVM, IBM System i™ platform now comfortably supports Java and WebSphere® applications on a wide array of different server models: from entry size boxes to the huge enterprise systems.

This book provides in-depth information about setting Java and IBM WebSphere environments with new 32-bit JVM, tuning its performance, and monitoring or troubleshooting its runtime with the new set of tools.

Information in this IBM Redbook helps system architects, Java application developers, and system administrators in their work with 32-bit JVM in i5/OS.

Important: Despite the fact that this book targets i5/OS implementation, most information in this book applies to all IBM server platforms, where the new 32-bit JVM is supported.

The team that wrote this IBM Redbook

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

Aleksandr V. Nartovich is a Senior IT Specialist in the ITSO, Rochester Center. He joined the ITSO in January 2001 after working as a developer in the IBM WebSphere Business Components organization. During the first part of his career, Aleksandr was a developer in IBM AS/400® communications. Later, he shifted his focus to business components development on WebSphere. Aleksandr holds two degrees, one in computer science from the University of Missouri-Kansas City and the other in electrical engineering from Minsk Radio Engineering Institute.

Adam Smye-Rumsby joined IBM in 2000. He currently works as a Software Support Specialist for IBM in the UK, solving customer problems related to IBM WebSphere Portal and Lotus® Domino®. Previously he worked at IBM Hursley Laboratory in the UK testing IBM Java software development kits (SDKs) on a variety of platforms. He has also worked as a tester on IBM Web Services technology. Adam holds a Computer Science degree from the University of Portsmouth.

Paul Stimets is an Advisory Software Engineer at IBM, working in the Rochester Support Center. He supports Java and WebSphere Application Server on the System i platform. He has over 8 years of experience at IBM and has worked with Java on System i platform since its introduction in 1998 with IBM OS/400® V4R2.

George Weaver is a software engineer with the System i Technology Center at IBM in Rochester, MN. He has provided consultations, education, writings, Web content, and videos to help System i clients, software vendors, business partners and solution providers enhance their applications and services portfolios over the last 13 years. His current areas of focus are Web application development, and WebSphere Application Server configuration, and administration, and performance.

Many thanks to Java developers at IBM Rochester, MN for their enthusiastic support of this IBM Redbook.

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

Bill Berg
Marc Blais
Arv Fisher
Steve Fullerton
Jesse Gorzinski
Sandra Marquardt
Mark Schleusner
Nishant Thakkar
Blair Wyman
IBM Rochester, Minnesota

Richard Chamberlain
Ben Corrie
Holly Cummins
IBM Hursley, UK

Become a published author

Join us for a two-week 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 will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts helps increase product acceptance and customer satisfaction. As a bonus, you will 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 IBM Redbooks™ to be as helpful as possible. Send us your comments about this or other IBM Redbooks in one of the following ways:

- Use the online **Contact us** review IBM Redbook form found at:

  ibm.com/redbooks

- Send your comments in an email to:

  redbooks@us.ibm.com

- Mail your comments to:

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

1

The world of Java

This chapter provides a technical overview of the world of Java. It describes the typical architecture of the Java platform and its components. Understanding the Java platform and the functionality of its components is the key to understanding the rest of the materials in this book.

This chapter includes the following sections:

1.1 Java: language and platform

Java has become one of the buzz words of the IT world. Not mentioning Java in the list of skills or in your business presentation is a bad form. Quite often when people say “Java” they mean many different things. And if you ask them to define more accurately “Java” in the context of their speech, not all of them give you the correct answer.

Because this is a technical book, it provides a brief introduction to several key definitions related to *Java*.

1.1.1 Java language

Java language is one of the most popular programming languages today. It has been around for many years since Sun™ Microsystems™ has introduced it in early 1990s.

Any programming language is based on a well defined *syntax* and *semantic*[1]. Any high-level language program, including Java, is a text file with a unique file extension (.java for the Java programs). You cannot execute a program on a computer as a text file. There is a requirement for another component, called *compiler*, that converts human readable text file into machine instructions. Typically, these instructions are unique for a specific hardware platform, such as Intel® or IBM PowerPC®. There is a compiler for Java too, but it works differently from other compilers. Refer to 1.2, “Java Virtual Machine” on page 4 and 1.4, “Java platform development tools and API” on page 8 for further information.

With time, people have noticed that some programming segments are duplicated in many different applications. As the result, the most frequently used functions (segments of programs) are prewritten and grouped in special libraries. These libraries are called differently for different platforms and languages. For Java they are called *Java class libraries*. You can download these packages to a computer and include (or reference) these packages to any program. Therefore you achieve *code reuse*.

As a language (meaning syntax and semantic) Java has not changed much, but the runtime support and the number of available packages have grown tremendously. As a result, Java programming language is one of the most popular languages for enterprise applications.

The main characteristics of the Java language includes the following:

- Java is a high-level programming language.
- Java is an *object-oriented* language.

  Many programming languages support this concept. It is based on the idea that the program is seen as the collection of *objects*. Each object represents an entity within the business task that this program has to solve. For example, many applications have Customer or Address objects. The objects communicate through well-defined interfaces, called *methods*, by exchanging messages.

  The object-oriented programming claims to achieve a better flexibility, reusability of the code, and maintainability. It is a popular choice for the large-scale solutions.
- Java is platform independent.

  A Java program written according to the best practices can be moved to any supported platform without any change or requirement to recompile.

[1] Semantic is the correct set of actions (or machine instructions) executed for each operand in your program.

1.1.2 Java platform and Java software development kit

To support the development and deployment of the Java applications, you must have a special environment on your system. This environment is called *Java platform*. Figure 1-1 shows the architecture of the Java platform. This chapter explores this architecture in more details in the following sections.

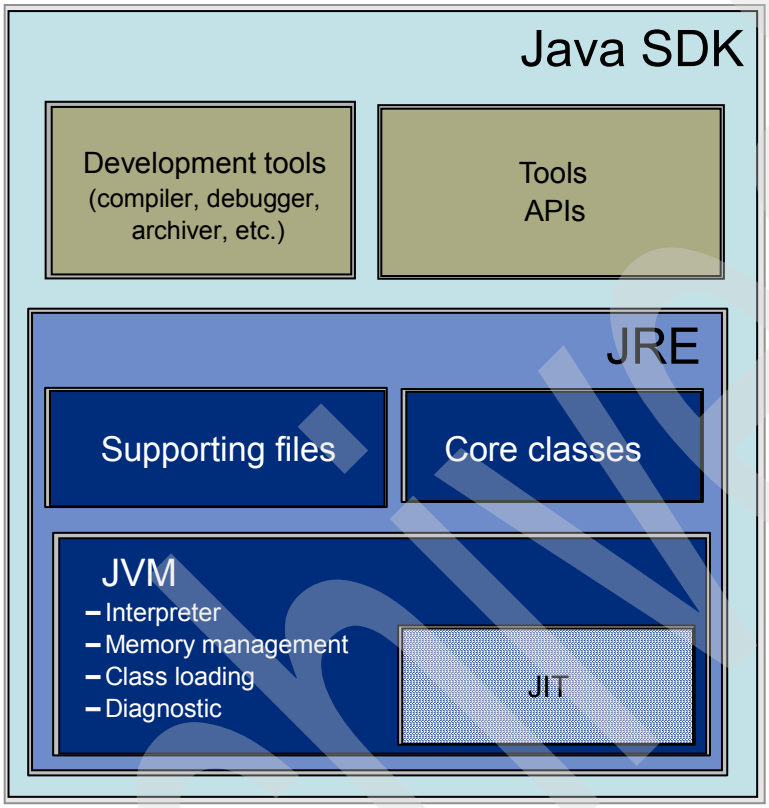


Figure 1-1 Java platform architecture

Different applications might require different Java platforms. Therefore, there are several editions of the Java platform in existence today:

- Java Platform Micro Edition (Java ME, formerly called J2ME™): This edition targets development and deployment of the Java applications for small, mobile devices, such as PDAs and cell phones.
- Java Platform Standard Edition (Java SE, formerly J2SE™): This edition targets development and deployment for most of the Java applications running on the servers.
- Java Platform Enterprise Edition (Java EE, formerly J2EE™): This edition has additional support application programming interface (API) for enterprise-level applications. The examples of additional API include Web services, distributed deployment of the applications, and communication API.

Java platform is a *document* which is formed of a *set of specifications*. These specifications define the services and APIs for the Java environment. The product, that implements this document, is called *Java software development kit* (Java SDK). It includes the following major components (see Figure 1-1):

- Java Virtual Machine (JVM)
- Java Runtime Environment (JRE™)
- Development tools

1.2 Java Virtual Machine

After you create a Java program, you have to compile this program. The result of this compilation is the file with extension *.class*. It contains *Java bytecode*. The generated bytecode is identical no matter on what platform you have compiled your program. It is platform neutral. That is why we can say that Java program is *platform independent*.

However, you cannot execute this bytecode yet because your system understands machine instructions that are unique to your system's architecture (PowerPC, Intel, and so on). In order to execute a Java program, you have to install a middleware component that fits between your Java program and operating system. This component is called *Java Virtual Machine*. JVM runs as an application on top of an operating system, as shown in Figure 1-2, and provides a runtime environment for all Java applications. JVM effectively emulates an operating system environment for Java programs. That is why it is called a *Virtual Machine*.

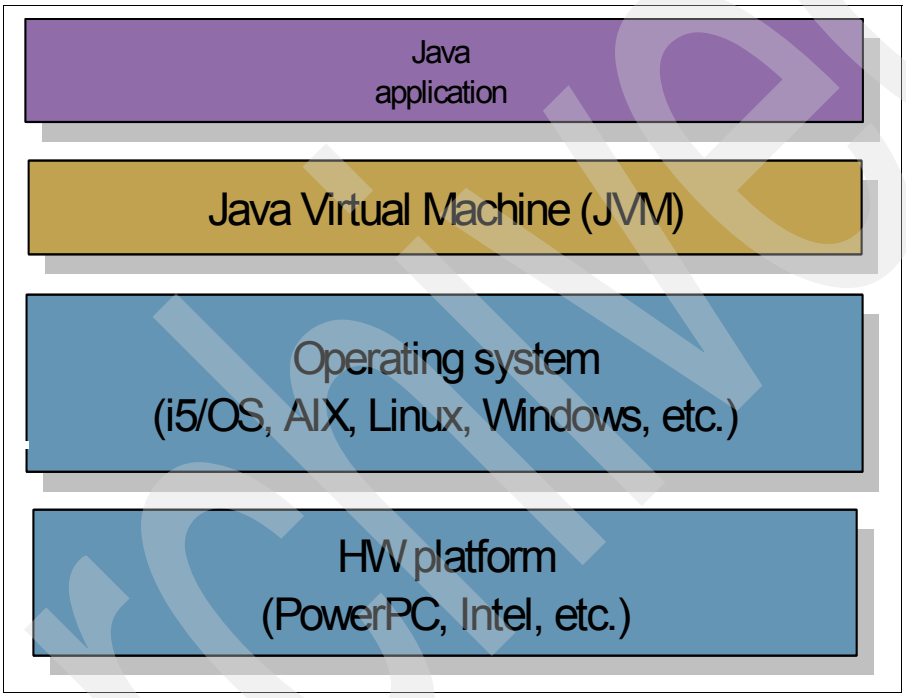


Figure 1-2 Java Virtual Machine

The main purpose of JVM is to convert the Java bytecode to the machine instructions that you can execute on the hardware platform where you run your application. JVM is the overhead in a Java Runtime Environment compared with traditional programming languages, such as C or RPG. However, this is the price you pay for Java portability.

Converting Java bytecode is not the only task that JVM performs. The next section looks at the JVM architecture, because a good understanding of the JVM components can help you in tuning, maintaining, and monitoring a JRE.

1.2.1 JVM architecture

Figure 1-3 demonstrates the typical components of JVM.

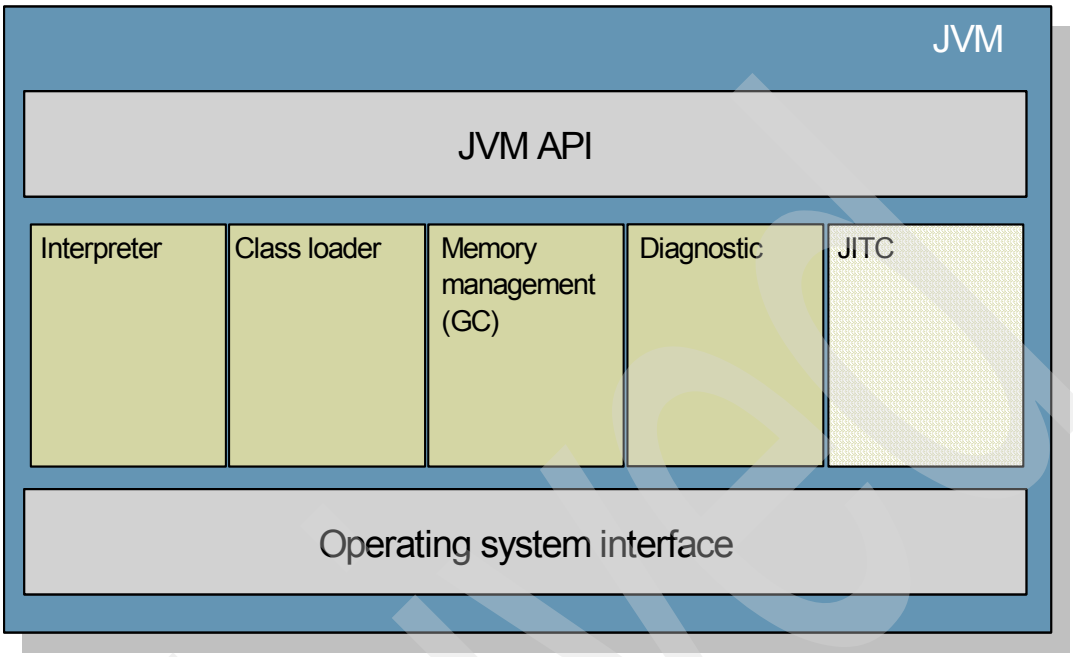


Figure 1-3 JVM architecture

Interpreter

Interpreter is a computer program that processes Java bytecodes by calling a fixed set of native instructions for each kind of Java bytecode. This is the slowest way to process Java bytecodes.

Class loader

Before Java interpreter can start converting Java bytecode, the Java class file has to be *loaded*. Java class loader is responsible for supporting Java's dynamic code loading facilities, which include the following:

- Reading standard Java .class files.
- Resolving class definitions in the context of the current runtime environment. Java language allows you to define more than one class with the same name. However, these classes have to be in the different packages[2]. Class loader is responsible for the correct class name resolution.
- Verifying the bytecodes defined by the class file to determine whether the bytecodes are language-legal.
- Initializing the class definition after it is accepted into the managed runtime environment. Class initialization includes memory allocation for the class and assignment of the initial values to all class parameters and variables.
- Supporting various reflection APIs for introspection on the class and its defined members.

[2] You can have multiple class loaders, and two versions of the same class in the same package can exist in the same JVM as long as they are loaded with two different class loaders.

Memory management

The memory management subcomponent is responsible for the efficient use of system memory by a Java application. Java programs run in a managed execution environment. When a Java program requires storage, the memory management subcomponent allocates a discrete region of unused memory called *Java heap* (or just *heap*). During runtime a Java application creates objects in the Java heap. After the application no longer refers to an object (the storage on the heap), the memory management subcomponent must recognize that the storage is unused and reclaim the memory for subsequent reuse by the application or return it to the operating system. This function of the memory management component is called *Garbage Collection* (GC).

For further information about memory management subcomponent, refer to Chapter 5, “Tuning the garbage collector” on page 35.

Diagnostics component

The diagnostics component provides *Reliability, Availability, and Serviceability* (RAS) facilities to the JVM. The IBM Virtual Machine for Java is distinguished by its extensive RAS capabilities. It is designed to be deployed in business-critical operations and includes several trace and debug utilities to assist with problem determination. If a problem occurs in the field, it is possible to use the capabilities of the diagnostics component to trace the runtime function of the JVM and help to identify the cause of the problem. The diagnostics component can produce output selectively from various parts of the JVM and the just-in-time (JIT) compiler.

JVM API

The JVM API encapsulates all the interaction between external programs and the JVM. Examples include the following:

- Creation and initialization of the JVM through the invocation APIs. Creating a JVM involves native code. JVM API provides *Java Native Interface* (JNI) API to communicate with the native code.
- Handling command-line directives. Any command line arguments that you supply during JVM startup are processed by JVM API component.
- Presentation of public JVM APIs such as JNI and *JVM Tool Interface* (JVMTI). JVMTI supports monitoring and debugging interfaces for the JVM tools.
- Presentation and implementation of private JVM APIs used by core Java classes.

Operating system interface

This interface provides the abstraction for the other components of JVM from the underlying platform. It allows other JVM components to be written in platform neutral manner. Therefore, you have to replace only operating system interfaces when a JVM vendor, such as IBM, creates JVM for different platforms.

Just-in-time compiler

Java bytecode interpretation is done by the JVM. It interprets one bytecode at a time. If your Java program calls the same method twice, the JVM is going to interpret that method two times. This is just one example. If you look at most of the business programs, string manipulations, math calculations, and database access methods are among the most typical operations in such applications.

However, there is a better way to handle interpretation processes of the most frequently used methods. This is the optimization mechanism that improves the overall performance of Java applications. This mechanism is implemented in the additional component called *just-in-time compiler*.

In simple terms the JIT compiler mechanism works as follows:

1. JIT compiler compiles the entire method before it is executed on the system.
2. JIT compiler saves the compiled version of the method on a heap, called *JIT heap*.
3. When the same method is called again, JIT compiler checks its heap and if the method is there, it runs without recompilation.

   Due to the physical limitations of the main memory, not all compiled methods can be stored in the JIT heap.

> **Note:** JIT compiler is not a required component of JVM, but it is found in all production implementations of JVM.

JIT compiler optimization in IBM Technology for Java Virtual Machine

If you try to analyze the usage pattern for all methods executed in JVM, you must see that some methods are executed a few times and some are executed tens of thousands of times over the JVM lifetime. It will be nice to keep only the most frequently-used methods on the JIT heap. If we could implement this technique, JIT compiler would greatly improve the chances of finding the compiled version of the method on the heap. But how does JIT compiler know what are the most frequently-used methods?

The following algorithm implements this optimization:

1. When JVM starts, it takes all methods through the interpreter, not JIT compiler.
2. At the same time, JVM counts how many times each method has been interpreted.
3. When this count reaches a predefined number N, it is called *JIT threshold*. JIT compiler compiles this method and places it on the *JIT heap*. The count of this method is reset.
4. Subsequent calls to the same method continue to increment its count. When the call count of a method reaches a *JIT recompilation threshold*, the JIT compiles it a second time, this time applying a larger selection of optimizations than on the previous compilation, because the method has proven to be a significant part of the whole program. The recompilation process is iterative. The call count of a recompiled method is reset again and, as it reaches succeeding thresholds, it triggers recompilation at increasing optimization levels. Thus, the busiest methods of a Java program are always optimized most aggressively, maximizing the performance benefits of using the JIT.
5. This process continues during the lifetime of the JVM.

1.3 Java Runtime Environment

Java Runtime Environment provides all necessary components to run Java applications on a specific platform. In i5/OS you get JRE as part of Java SDK.

Figure 1-1 on page 3 shows the main components of JRE:

- JVM

  This has been described 1.2, “Java Virtual Machine” on page 4.

- Core classes

  As described in 1.1.2, “Java platform and Java software development kit” on page 3, Java platform has three editions. Each edition includes a predefine set of APIs. Core classes are the compiled classes for all APIs that are supported by a specific edition of Java platform. They usually come in a form of Java archive (JAR) files.

- Supporting files

  This includes multiple property files and additional tools. For example, there are two executable programs that allow you to support Remote Method Invocation (RMI) of Java methods. On the other hand, there is the Java Control Panel tool that provides the method to view and update some of the global parameters related to JRE.

Note: You get JRE in i5/OS by installing 5722JV1 option 8 (J2SE 5.0 32-bit). JRE is located in the following directory in the Integrated File System (IFS):

```
/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/
```

Unlike other platforms, there is no separate install for JRE on i5/OS.

1.4 Java platform development tools and API

Java SDK includes JRE and development tools and APIs. The following tools are the most important tools in SDK:

- *javac*: Java compiler. It produces the Java bytecodes.
- *jar*: Java archive. You can archive multiple files and packages in a single JAR file for ease of use and maintenance. You can package your entire application in a JAR file.
- *javadoc*: The tool that creates the Java documentation based on the comments in the source code.
- *jdb*: This is a Java debugger.
- *idlj*: This is the Interface Definition Language (IDL) to Java compiler. It generates the Java bindings based on an IDL file. It is used to allow Java programs to participate in Common Object Request Broker Architecture (CORBA).

In addition to the tools, Java SDK includes examples of Java programs and development APIs, such as internationalization and IDL API.

2

Exploring IBM Technology for Java Virtual Machine

This chapter discusses how Java Virtual Machine (JVM) technology has evolved on the IBM System i platform. This chapter includes the following:

2.1 A history of Classic JVM and IBM Technology for JVM on System i platform

The Java platform has been available on System i platform since i5/OS (OS/400 at the time) V4R2, released in 1998. Prior to direct Java support in i5/OS V4R2, IBM delivered the IBM Toolbox for Java. The IBM Toolbox for Java is a library of classes that enable Java applications to directly access System i resources such as database files, data queues, data areas, and callable program objects. The IBM Toolbox for Java provided the conduit for a whole new set of client/server and host-based business applications to utilize System i resources.

IBM also released the IBM Toolbox for Java to the open source community as JTOpen. This enabled Java application developers to create applications that access System i resources, even if they did not have a System i platform in their IT infrastructure. IBM continues to enhance both the licensed program product version (5722-JC1) and the JTOpen versions of IBM Toolbox for Java.

At that time one of the most common usages of Java with System i applications was Java applets. A Java applet is a Java application that typically runs within the JVM support included with a Web browser. A Java developer would create an applet that, for example, displayed a bar graph of data. A Web developer would code a Hypertext Markup Language (HTML) page to launch this applet within the user's browser session.

More importantly from a System i perspective, Java applications could run directly on i5/OS. Application developers could take advantage of the "write once run anywhere" capabilities that Java afforded and make them available on multiple platforms, including System i platform. Workstation-based applications that used the IBM Toolbox for Java classes to access System i resources could run directly on System i platform.

2.1.1 The System i Classic JVM

IBM System i developers were tasked with designing and developing a Java platform for i5/OS that would not only just run Java bytecode classes according to Sun specification, but also take advantage of the ease of use, scalability, reliability, and security that the System i clients had enjoyed for many years. Another key design requirement was optimizing performance. At the time, Java applications were interpreted and therefore had a tendency to consume more processor and memory resources than compiled code.

System i platform uses a technology-independent machine interface (TIMI) to separate the i5/OS operating system and applications from the underlying hardware implementation. IBM developed a set of new instructions below the machine interface specifically to support Java. The actual Java code and application programming interface (API) were (and continue to be) implemented above the machine interface (licensed program product 5722-JV1 on i5/OS releases) as shown in Figure 2-1.

See Figure 2-1.

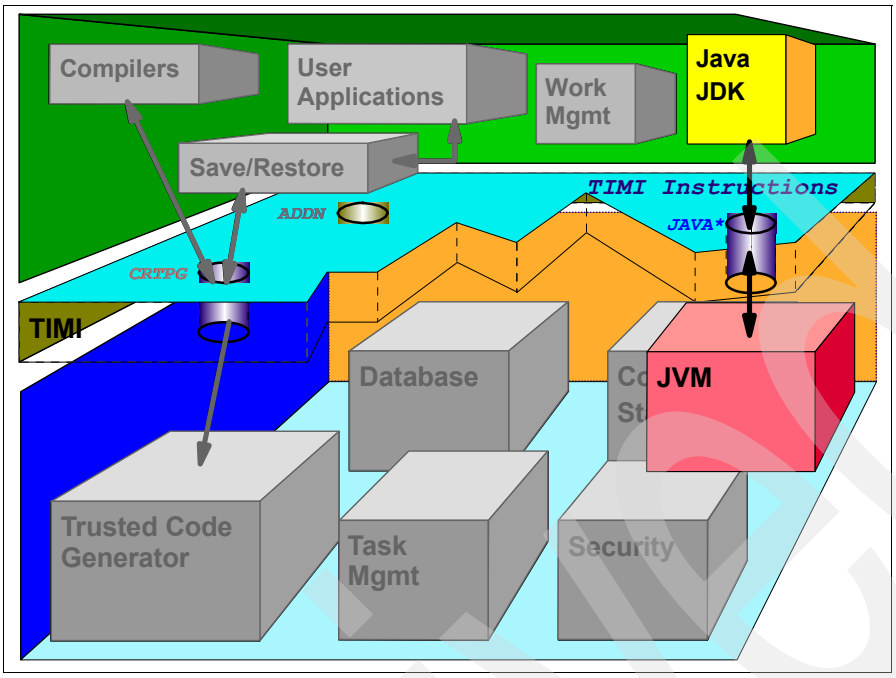


Figure 2-1 Classic Java platform implementation on System i platform

i5/OS provided interfaces to the Java platform via the Qshell environment. i5/OS also provided a RUNJVA control language (CL) command that enabled Java applications to be called directly from a command line or CL program. i5/OS also included a CRTJVAPGM CL command to optimize application performance and create a reusable program object that could be saved between system initial program loads (IPLs). See Figure 2-2. This is known as the direct executable (DE) environment.

```
                    Create Java Program (CRTJVAPGM)

Type choices, press Enter.

Class file or JAR file . . . . . > '/home/myfolder/Hello.class'

Classpath  . . . . . > '.:/home/myfolder'

Java developer kit version . . .   *NONE         Character value, *NONE
Optimization . . . . . > 30        10, *INTERPRET, 20, 30, 40
User profile . . . . . .   *USER         *USER, *OWNER
Use adopted authority  . . . . .   *NO           *NO, *YES
Replace program  . . . . . .   *YES          *YES, *NO
Enable performance collection  .   *NONE         *NONE, *ENTRYEXIT, *FULL
Profiling data . . . . . .   *NOCOL        *NOCOL, *COL

                         Additional Parameters

Directory subtree  . . . . . .   *NONE         *NONE, *ALL
Target release . . . . . .   *CURRENT      *CURRENT,V4R4M0, ...
                                                                    More...
```

Figure 2-2 i5/OS CRTJVAPGM code screen

Java on i5/OS was able to take advantage of the System i 64-bit architecture to provide a scalable solution from single processor machines all the way up to multiprocessor machines. The Classic JVM was unique in its implementation of asynchronous garbage collection, which allowed the JVM to continue processing application requests during the garbage collection cycle. As Java usage requirements evolved from primarily stand-alone applications to distributed, Web based applications, IBM has evolved the Classic Java platform environment as well. WebSphere Application Server is an excellent example. Very early versions of WebSphere Application Server made extensive use of directly executable programs on System i platform, because this typically provided better performance than interpreted code.

Developments in JVM technology and just-in-time (JIT) compilers continued to be added to the Classic JVM, especially in conjunction with IBM WebSphere Application Server. In i5/OS V5R2, IBM introduced a mixed mode interpretation (MMI) environment where each Java method has been interpreted a finite number of times (2000 by default). After this interpretation threshold has been reached, the method is compiled into native code by the JIT compiler. In recent releases, and especially for WebSphere Application Server based workloads, JIT compiled methods have tended to perform better than the DE environment.

The Classic JVM is used by default for Java applications run from the i5/OS Qshell environment, and the RUNJVA CL command is also used. The Classic JVM is also the default for WebSphere Application Server workloads. IBM releases new versions of the Java Development Kit (JDK™) by creating new options for the base IBM Developer Kit for Java licensed program product. Fixes are delivered by the standard System i program temporary fix (PTF) process. IBM provides a Java group PTF that includes fixes for the specific i5/OS release (which is V5R4M0) and the software development kit (SDK) versions (which are 1.3, 1.4, and 5.0). This greatly simplifies system management tasks, especially compared with other platforms that require a separate fix pack installation for each JDK version.

2.1.2 IBM Technology for Java Virtual Machine

1.1.2, “Java platform and Java software development kit” on page 3 discusses the different Java editions: ME (for small handheld or micro devices), SE (for standard workstation and server applications), and EE (for distributed and transactional applications). The System i Classic JVM is based upon the standard edition, similar to the SDKs for other platforms and operating systems. Creating different JVM implementations for different Java editions and operating systems is expensive and time consuming. It can also make it difficult to leverage improvements across the different implementations.

IBM has invested very heavily in Java technologies for over 10 years. IBM research has recently developed a highly configurable core JVM that you can use for micro edition implementations, where a small footprint is of utmost importance, and you can also use it for server and enterprise implementations, where scalability and extensibility are necessary. This JVM also has configurable garbage collection policies, which are discussed in further detail in 5.2.1, “Garbage collection policies” on page 37, and advanced performance profiling and optimization features. This new JVM implementation is referred to as IBM Technology for JVM throughout the rest of this book.

IBM Technology for JVM is available and supported on most of the popular operating systems, including i5/OS. IBM Technology for JVM is compliant with JSE version 1.4.2 and version 5.0 specifications. IBM Technology for JVM is available on i5/OS V5R4M0 and later releases as 5722-JV1, option 8, and is compliant with JSE version 5.

2.2 Comparing IBM Technology for JVM and Classic JVM

This section discusses about the similarities and differences between IBM Technology for JVM and Classic JVM. This provides the foundation to help you determine if the IBM Technology for JVM option is suitable for your System i environment, and if it is more advantageous than Classic JVM.

2.2.1 Similarities

Before discussing the differences between Classic JVM and IBM Technology for JVM for System i platform, it is helpful to understand the similarities. The most important similarity is that both VMs are fully compliant with the JSE specifications, and Java programs can run in either VM without modification or recompilation. You can use both environments within the i5/OS Qshell environment to compile (the **javac** command) and run (the **java** command) Java programs. Both are shipped as part of the 5722-JV1 (IBM Developer Kit for Java).

Both can use the RUNJVA CL command, although some parameters for IBM Technology for JVM are ignored. Both can use environment variables defined by the i5/OS ADDENVVAR CL command. Both have their fixes bundled with the i5/OS Java group PTF (SF99291 for V5R4M0). Both can use the IBM Toolbox for Java to access resources such as data queues and data areas. Both can use the System i native Java Database Connectivity (JDBC™) driver to access database resources. Both can use System i debugging API's that are started via STRSRVJOB or STRDBG CL commands.

2.2.2 Differences

There are differences between IBM Technology for JVM and Classic JVM. Some are based upon how IBM Technology for JVM is implemented on System i platform. Others are based on the different code bases used between the two.

System i platform implementation differences

The Classic JVM is implemented primarily below the System i TIMI and is available on all currently supported i5/OS releases. The IBM Technology for JVM is implemented in the i5/OS portable application solutions environment (PASE) and is currently available on i5/OS V5R4M0, as shown in Figure 2-3.

PASE is discussed in more detail in 3.1, "i5/OS portable application solutions environment" on page 20. The PASE on System i platform does not use an emulation or operating system layer and tends to be very fast. The PASE can access System i resources such as the file system, programs, and Transmission Control Protocol/Internet Protocol (TCP/IP) sockets. Therefore IBM Technology for JVM can access the same resources as Classic JVM.

See Figure 2-3.

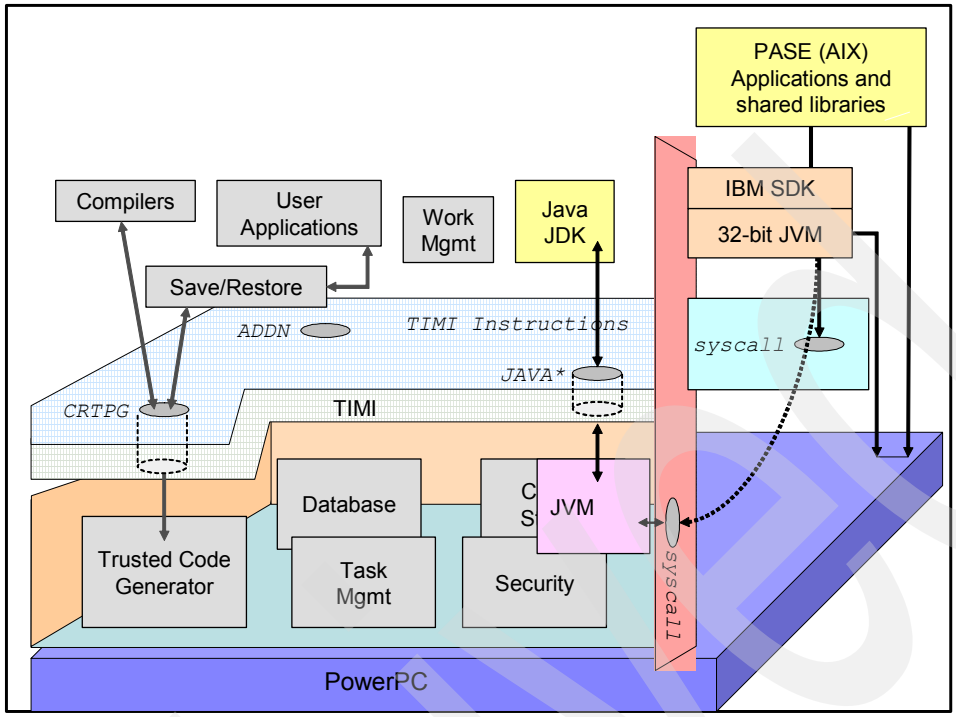


Figure 2-3 IBM Technology for JVM implementation on System i platform

The PASE on System i platform supports 32-bit and 64-bit address spaces. With i5/OS V5R4M0, IBM Technology for JVM currently uses a 32-bit address space. This limits the theoretical maximum memory usage of IBM Technology for JVM to approximately 4 GB (2^{32} bytes) versus approximately 17 billion GB (2^{64} bytes) for Classic JVM. The IBM Technology for JVM uses approximately 20% of the memory for its own internal operations, limiting the maximum heap to approximately 3.25 GB (the practical limit for most applications is in the 2.5 GB to 3.0 GB range). Classic JVM has a maximum heap of 243 GB.

Even though the IBM Technology for JVM's use of a 32-bit address space limits the maximum heap size compared with Classic JVM, this does have advantages. The 32-bit address space uses less memory for the JVM, compared with Classic JVM. As a rule of thumb, the heap size required for the IBM Technology for JVM is approximately 60% of the Classic JVM, all things being equal. Suppose you are running WebSphere Application Server workloads today with the Classic JVM and have a heap size of 4 GB. That same environment running with IBM Technology for JVM would be approximately 2.5 GB and most likely be able to run fine.

The IBM Technology for JVM is displayed as an application in the PASE. This environment is very similar to the implementation of IBM Technology for JVM in AIX®. It yields good performance because it is so close to the processor and has a relatively short code path. Because it is not implemented above the System i TIMI, it does not have implementation independence from the underlying hardware changes. However, this must only impact the JVM and not end user applications.

The i5/OS CRTJVAPGM and DSPJVAPGM commands are not applicable with the IBM Technology for JVM. Also, the RUNJVA command parameters such as OPTIMIZE are ignored.

Other differences

One of the biggest differences is in garbage collection algorithms available in IBM Technology for JVM. The Classic JVM uses a System i platform unique algorithm, and typically the only tuning parameter is the initial and maximum heap size (although most customers set the maximum heap size parameter to *NOMAX). The IBM Technology for JVM has several garbage collection algorithms and gives the option of setting the initial heap size and maximum heap size to an appropriate value. Many other options are available for fine-tuning GC behavior, which is a further differentiator compared to Classic JVM. Chapter 4, “Making the switch to IBM Technology for JVM” on page 27 covers this in more detail.

Another difference is that many applications must see performance improvements such as decreased response time and higher throughput. One reason is because of the smaller memory requirements: object references are 32 bits with IBM Technology for JVM, compared with 64 bits for Classic JVM. Also, IBM has optimized the IBM Technology for JVM for the PASE. The *calling* methods for applications running in IBM Technology for JVM that utilize System i services (such as Integrated Language Environment® (ILE) programs and TCP/IP sockets) incur a slight increase in overhead compared to Classic JVM. However, the *execution* methods within applications running in IBM Technology for JVM incur less overhead than with Classic JVM and typically more than compensate for the calling methods.

Finally, there are substantial differences in monitoring agents and diagnostic tools. Both VMs support a set of tools. However, in most cases these tools are different for each VM. For System i developers, who have been working with the native tools, Table 2-1 shows the support of native tools for IBM Technology for JVM.

Table 2-1 JVM performance tools compatibility

Tool	Usage with IBM Technology for JVM
ANZJVM command	No
DMPJVM command	No
Service tools JavaGCInfo macro	No
iDoctor for IBM iSeries™ Heap Analysis Tools for Java	No
Performance Explorer	Some
Tivoli® Performance Viewer (WebSphere Application Server)	Yes

The new set of tools for IBM Technology for JVM is described in Chapter 8, “Analyzing JVM behavior” on page 83.

There are a few other differences that are listed in 2.4, “Using IBM Technology for JVM with existing applications” on page 16.

2.3 Choosing IBM Technology for JVM or Classic JVM

Choosing the appropriate Java version 5 SDK must not be a difficult exercise. System i platform gives you the flexibility to use IBM Technology for JVM and Classic JVM. You can try both for your specific applications and System i environment, and choose the better performer, considering lower response time, more throughput, and so on.

However, if you are using i5/OS V5R3 you do not have the option to use IBM Technology for JVM. WebSphere Application Server version 6.0 and earlier releases do not use Java version 5, therefore the Classic JVM is your only option for pre-WebSphere Application Server 6.1 workloads, even if you are using i5/OS V5R4.

If you do have i5/OS V5R4 or later and, optionally, WebSphere Application Server version 6.1 or later, then how do you choose? If scalability in a single JVM process on a large multiprocessor System i platform (which means eight or more processors) or i5/OS integration are important, then Classic JVM might be a better fit.

In general, you must estimate the memory requirements for your application running in IBM Technology for JVM. If the application's memory requirements fit in 32-bit addressable space, you must test your application in both VMs to choose the best performing one. Refer to Chapter 4, "Making the switch to IBM Technology for JVM" on page 27 for more details.

In our tests IBM Technology for JVM, on average, has been 7-10% faster than Classic JVM.

2.4 Using IBM Technology for JVM with existing applications

Generally speaking, applications that run within Classic JVM (5722-JV1, option 7) must run within IBM Technology for JVM (5722-JV1, option 8). For example, WebSphere Application Server version 6.1 is the first version that runs on the Java version 5 SDK. This is supported on Classic JVM and IBM Technology for JVM, on i5/OS V5R4M0. There are a few caveats you must be aware of:

- Applications running within IBM Technology for JVM do not support i5/OS adopted authority. Refer to 7.3.4, "Adopted authority" on page 77.
- ILE C programs that are linked must be compiled with the teraspace option on (STGMDL *TERASPACE), if used with IBM Technology for JVM.
- PASE 64-bit address space based applications are not supported on IBM Technology for JVM.

2.5 Future JVM developments on i5/OS

IBM has, and continues to make, substantial investments in Java technology. At the same time, IBM also understands your requirements to preserve your investment in applications and skills, and to implement new technologies according to your own timetable. Accordingly, the Classic JVM will still be supported in the next i5/OS release. This gives you the option of evaluating and testing your applications on IBM Technology for JVM, at a pace suitable for your organization.

IBM Technology for JVM represents IBM strategic JVM offering. Using a common code base across multiple platforms provides a focus for IBM research and development investment in Java technology.

In the future you can expect to see a 64-bit version of the IBM Technology for JVM. This enables you to compare the 32-bit and 64-bit versions of the IBM Technology for JVM to determine which is optimal for your specific environment. The eventual introduction of 64-bit IBM Technology for JVM on i5/OS brings i5/OS in line with other platforms such as AIX and Linux®, for which both 32-bit and 64-bit versions of IBM Technology for JVM are available.

You can expect to see performance improvements, and also improved monitoring and analysis tools as the IBM Technology for JVM continues to mature. Some of these tools are available on multiple platforms. Some may be System i platform unique.

Finally, you can expect to see application development tools such as WebSphere Development Studio Client for iSeries, incorporate IBM Technology for JVM. This will enable developers to extensively test and debug their stand-alone applications and WebSphere Application Server based IBM Technology for JVM applications prior to putting them into production on a System i platform.

3

New user guide

This chapter discusses how you can start using the IBM Technology for Java Virtual Machine (JVM). This chapter covers the following topics:

3.1 i5/OS portable application solutions environment

As mentioned in Chapter 1, “The world of Java” on page 1 and Chapter 2, “Exploring IBM Technology for Java Virtual Machine” on page 9, the IBM Technology for Java Virtual Machine is targeted to be the IBM main virtual machine for Java, based on a common, high-speed, modular code base.

To maximize the reuse of common code, i5/OS uses the AIX version of the IBM Technology for JVM as its base. The code is customized for use under the i5/OS portable applications solutions environment (i5/OS PASE). Because most of the code is common, the i5/OS version is able to rapidly adapt updates and fixes made in IBM Technology for JVM, especially those updates with AIX-specific benefits or considerations.

3.2 Installing IBM Technology for JVM

A successful install involves several steps. This section discusses the necessary prerequisites, the installation, and verification.

3.2.1 Checking prerequisites

To ensure a successful installation of the IBM Technology for JVM you must make sure that your system meets the following prerequisites:

- Make sure that your system is running i5/OS V5R4. If your system is running a version of i5/OS that is older than V5R4 you have to upgrade before you can use IBM Technology for JVM.
- Your system must also have i5/OS PASE installed. To verify that i5/OS PASE is installed perform the following steps:
  a. Enter GO LICPGM command.
  b. Select **10** (Displays installed licensed programs).
  c. Press F11 until you see the column *product option* open.
  d. Page down and look for option 33 (Portable App Solutions Environment) of 5722SS1 as shown in Figure 3-1.

  If you find that i5/OS PASE is *not* installed, review the iSeries Information Center, V5R4 for installation instructions on the Web at:

  `http://publib.boulder.ibm.com/infocenter/iseries/v5r4/topic/rzalf/rzalfgetstart.htm`
- Ensure that the 5722JV1 product with option *BASE is installed on your system

It is also important to ensure that the latest program temporary fixes (PTFs) are installed. i5/OS requires:

- Latest cumulative PTF package
- Latest HIPER Group PTF (SF99539)

> **Note:** For more information about required PTFs, visit the following Web site:
>
> `http://www.ibm.com/systems/support/i/fixes/index.html`

See Figure 3-1.

```
                    Display Installed Licensed Programs
                                                          System:   RCHAS60
Licensed  Product
Program   Option   Description
5722SS1    18      Media and Storage Extensions
5722SS1    21      Extended NLS Support
5722SS1    22      ObjectConnect
5722SS1    23      OptiConnect
5722SS1    25      NetWare Enhanced Integration
5722SS1    26      DB2 Symmetric Multiprocessing
5722SS1    27      DB2 Multisystem
5722SS1    29      Integrated Server Support
5722SS1    30      Qshell
5722SS1    31      Domain Name System
5722SS1    33      Portable App Solutions Environment
5722SS1    34      Digital Certificate Manager
5722SS1    35      CCA Cryptographic Service Provider
5722SS1    36      PSF 1-45 IPM Printer Support
                                                                   More...
Press Enter to continue.

F3=Exit   F11=Display status   F12=Cancel   F19=Display trademarks
```

Figure 3-1 Verifying that i5/OS PASE is installed

For i5/OS PASE, the list of required PTFs can be found here:

http://www-03.ibm.com/servers/enable/site/porting/iseries/pase/misc.html

3.2.2 Installing IBM Technology for JVM

Installing the IBM Technology for JVM product is a simple process.

IBM Technology for JVM is included in licensed program 5722-JV1 (option 8). The install media for 5722-JV1 is included on the V5R4 Standard Set Media. Option 8 of 5722-JV1 is typically found on the CD labeled *D29xx_07*.

The following steps install IBM Technology for JVM:

1. Make sure the install CD is in the System i optical drive.
2. Start the installation with the following command:

   ```
   RSTLICPGM LICPGM(5722JV1) DEV(OPT01) OPTION(8)
   ```

   This is assuming OPT01 is the name of your optical device.

When the installation completes you have to install the latest Java group PTF (SF99291). An easy way to install SF99291 is by using the GO PTF menu and take option 8 (install program temporary fix package). When this book was written, SF99291 was at level 4. You can use the following Web site to check if there is a more current version:

http://www.ibm.com/systems/support/i/fixes/index.html

On a CL command line you can use the WRKPTFGRP command to verify the level of each group PTF installed on the system.

3.2.3 Verifying the installation

With the PTFs installed, the final step is to verify that the new product is functional. Run the following commands:

1. Open a Qshell session:

   ```
   STRQSH
   ```

2. Add JAVA_HOME to your shell environment:

   ```
   export JAVA_HOME=/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit
   ```

3. Invoke the Hello program included with the Java product (5722JV1) to verify the installation:

   ```
   java -showversion -classpath /qibm/proddata/java400 Hello
   ```

   Figure 3-2 shows the example of running this command. Notice that the dates for the JVM component builds may be different when you run this program on your system.

   Note: By adding **`-showversion`** to the **`java`** command additional lines of output are displayed that indicate which JVM is in use.

```
                              QSH Command Entry

$
export JAVA_HOME=/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit
$
java -showversion -classpath /qibm/proddata/java400 Hello
java version "1.5.0"
Java(TM) 2 Runtime Environment, Standard Edition (build jclap32dev)
IBM J9 VM (build 2.3, J2RE 1.5.0 IBM J9 2.3 OS400 ppc-32  (JIT enabled)
J9VM - 20060501_06428_bHdSMR
JIT  - 20060428_1800_r8
GC   - 20060501_AA)
JCL  - jclap32dev

Hello World
$
```

Figure 3-2 Running the Hello program to verify IBM Technology for JVM installation

If your system has a java.version=1.3 or 1.4 specified in a property file, you may still see an output for Classic JVM. Adding the property `-Djava.version=1.5` on a command line assures that you actually use IBM Technology for JVM, for example:

```
java -showversion -Djava.version=1.5 -classpath /qibm/proddata/java400 Hello
```

3.2.4 Uninstalling IBM Technology for JVM

If you have to remove the IBM Technology for JVM for any reason, you can use the following command:

```
DLTLICPGM LICPGM(5722JV1) OPTION(8)
```

3.3 Basic configuration

This section discusses how to set up your environment to use IBM Technology for JVM. If both Classic JVM and IBM Technology for JVM are installed, the default behavior on i5/OS V5R4 is to use Classic JVM.

3.3.1 Setting JAVA_HOME

As mentioned in 3.2.3, “Verifying the installation” on page 22 we set JAVA_HOME before running the Hello Java program. Setting JAVA_HOME to the installation directory of IBM Technology for JVM toggles between Classic JVM and IBM Technology for JVM.

In order to activate the IBM Technology for JVM, JAVA_HOME must be set to the following:

```
/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit
```

Note: This value is case sensitive. If JAVA_HOME is set to an invalid directory, any attempt to invoke Java fails with an error similar to Figure 3-3.

If you are familiar with the Classic JVM on i5/OS you know that you can have multiple JDKs installed (1.3, 1.4, 1.5, and so on) and you can switch between them by setting the **`java.version`** property. The only way to use IBM Technology for JVM is by setting the JAVA_HOME environment variable. The behavior of JAVA_HOME is as follows:

- If JAVA_HOME is not set, or set to the empty string, all forms of Java invocation uses the i5/OS default Classic JVM implementation. If the Classic JVM is not installed, IBM Technology for JVM is used.
- If **`java.version`** is set to 1.3 or 1.4, JAVA_HOME is ignored and Classic JVM is used.
- If JAVA_HOME is set to a valid JVM installation directory, all Java invocations use the specified VM.
- If JAVA_HOME is set to any other value, Java invocation fails with an error message as shown in Figure 3-3.

```
                              Java Shell Display

JAVA_HOME directory /BLAH not found. Java Virtual Machine not created.
Java program completed with exit code 1
```

Figure 3-3 Showing the error message displayed when JAVA_HOME is not set correctly

Three techniques you can use to set JAVA_HOME

Perform the following steps:

1. Use the ADDENVVAR CL command. For example, to add the variable to your current interactive job enter the following on a CL command line:

   ```
   ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit’)
   ```

   Use the following if you want to set the variable at the system level:

   ```
   ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32
   bit') LEVEL(*SYS)
   ```

Note: Use caution when setting JAVA_HOME to level *SYS. All subsequent Java invocations can be affected if they do not explicitly set JAVA_HOME in their respective startup routines.

If you want to remove the JAVA_HOME variable type, use the following:

```
RMVENVVAR JAVA_HOME
```

2. Set JAVA_HOME from within QSHELL. For example:
   a. Open a Qshell session using:

      ```
      STRQSH
      ```

   b. Export the variable to your process:

      ```
      export -s JAVA_HOME=/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit
      ```

   To remove the JAVA_HOME environment variable from within Qshell use the **unset** command. For example, use the following:

   ```
   unset JAVA_HOME
   ```

3. Use a *.profile* file to initialize the shell. If you would like JAVA_HOME to be set every time you start Qshell, you can create a *.profile* file in a user's home directory. For example, if your i5/OS user profile is *John* you would perform the following steps:
   a. Create a file called *.profile* in **/home/John**.
   b. Add the following text to the file:

      ```
      export -s JAVA_HOME=/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit
      ```

   c. Save the file.

   Now, when you enter Qshell it initializes the shell with JAVA_HOME. From within the Qshell session, run the **env** command to confirm that JAVA_HOME is set. See the sample output in Figure 3-4.

```
                              QSH Command Entry

$
env
LANG=/QSYS.LIB/EN_US.LOCALE
JAVA_HOME=/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit
QIBM_USE_DESCRIPTOR_STDIO=I
TRACEOPT=UNLINK
QIBM_DESCRIPTOR_STDERR=CRLN=N
QIBM_DESCRIPTOR_STDOUT=CRLN=N
QIBM_DESCRIPTOR_STDIN=CRLN=Y
LOGNAME=PAULS
SHLVL=1
HOSTTYPE=powerpc
HOSTID=123.123.123.123
HOSTNAME=RCHAS60.SERVER.IBM.COM
OSTYPE=os400
MACHTYPE=powerpc-ibm-os400
TERMINAL_TYPE=5250
...
```

Figure 3-4 JAVA_HOME is set correctly

3.3.2 Setting default command line options in SystemDefault.properties file

If you are familiar with using the Classic JVM, you have probably used the SystemDefault.properties file to set a Java system property or command-line option. It is an i5/OS specific mechanism and it also works for IBM Technology for JVM.

If you prefer to use a method that works on all platforms, you can refer to Appendix D of the *IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide*, available at:

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

You can create the SystemDefault.properties file in either of the following locations:

- Your user.home directory:

  `For example: /home/John/SystemDefault.properties`

- The /QIBM/UserData/Java400 directory:

  `/QIBM/UserData/Java400/SystemDefault.properties`

If you prefer to customize the name and location of the properties file you can define the QIBM_JAVA_PROPERTIES_FILE environment variable. The following is a sample of a CL command:

```
ADDENVVAR ENVVAR(QIBM_JAVA_PROPERTIES_FILE)
       VALUE(/qibm/userdata/java400/mySystem.properties)
```

It is certainly possible to have a SystemDefault.properties file in multiple locations. In that case the order of precedence is as follows:

1. Command line or JNI invocation API
2. QIBM_JAVA_PROPERTIES_FILE environment variable
3. The /home/<userid>/SystemDefault.properties file
4. /QIBM/UserData/Java400/SystemDefault.properties
5. Default system property values

For additional details and examples review Chapter 17 in the *IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide,* available at:

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

3.4 Known issues

There are a few known issues with running IBM Technology for JVM in i5/OS. See Table 3-1 for the details.

Table 3-1 Known problems

Problem	Cause	Possible Solution
java.class.path property and $CLASSPATH environment variable are ignored.	Classpath is being overridden by a java.class.path entry in a system properties file.	Modify system properties file accordingly.
IBM Technology for JVM does not run my PASE native method from Qshell. It fails with a java.lang.UnsatisfiedLinkError.	QIBM_JAVA_PASE_START UP environment variable is not set or is set improperly.	Set QIBM_JAVA_PASE_START UP to '/usr/lib/start32'.

Problem	Cause	Possible Solution
Output is unreadable on screen.	The os400.stdio.convert property may be set to 'N'.	Check properties files to make sure that os400.stdio.convert is not altered.
IBM Technology for JVM does not support adopted authority.	This is not a standard feature of a JVM.	Modify your program to remove the adopt authority dependency (refer to 7.3.4, “Adopted authority” on page 77).
Call to native program fails.	Program was not compiled to support teraspace.	Recompile the native program with the following options: TERASPACE(*YES) STGMDL(*TERASPACE) DTAMDL(*LLP64).
IBM Technology for JVM does not support calls to the 64-bit based programs in PASE.		
JAVA_HOME points to IBM Technology for JVM, but your Java program starts in Classic JVM.	java.version is set to 1.3 or 1.4 in one of the property files.	Change java.version to 1.5.

4

Making the switch to IBM Technology for JVM

This chapter provides information that helps you decide if switching to IBM Technology for JVM is right for your application. This chapter covers the following:

4.1 Fitting your application into a 32-bit JVM

When deciding to switch to the new IBM Technology for JVM you have to consider that you are moving from a 64-bit to a 32-bit implementation. This means that every reference to an object is half the size. This does *not* imply that your application now requires 50% less memory. Some early testing has shown that 20-50% reduction in heap (memory footprint) is a reasonable expectation. Therefore, now you must be able to make a quick calculation to see if your application is a candidate to fit in IBM Technology for JVM. A general rule of thumb is that any Classic JVM that is less than about 5 GB in size must run in IBM Technology for JVM. After you switch to the IBM Technology for JVM, be sure to test your application to ensure that any peaks in workload do not result in an OutOfMemoryError.

Tip: A general rule of thumb is that any application running under Classic JVM with a heap that is less than about 5 GB in size must run in IBM Technology for JVM without a problem.

The general procedure recommended for switching an application from Classic JVM to IBM Technology for JVM is as follows:

1. Measure the heap usage of the application under Classic JVM. Refer to 4.1.1, “Measuring heap usage in Classic JVM”. If the heap usage is significantly larger than 5 GB under Classic JVM, the application may run out of heap space under IBM Technology for JVM.

   If you are trying to run a new application, skip this step.

2. Estimate the required maximum heap size under IBM Technology for JVM based on the usage under Classic JVM. You can do this if you multiply your heap requirement in Classic JVM by 0.6.

   If you are testing a new application, or are not certain about the performance characteristics of an existing application running in the Classic 64-bit VM, start by running the application in IBM Technology for JVM with the default heap size parameters (currently an initial heap size of 4 MB and a maximum of 2 GB).

3. Determine heap settings based on estimated heap requirements. Refer to 5.2.2, “Tuning heap size” on page 41 for information about how to change heap settings.

4. Test application under load using IBM Technology for JVM, collecting verbose garbage collection (GC) data. Keep the default optthruput GC policy, but use the heap settings from step 3 listed previously.

   Refer to 5.2.3, “Verbose GC output” on page 44 for information about how to collect verbose GC data.

   If performance/memory usage do not meet expectations or OutOfMemoryErrors are generated, then perform the following steps:

   a. Analyze heap usage using EVTK or manual review of verbose GC data. Refer to “Troubleshooting garbage collection” on page 133 for information about using EVTK.

   b. Adjust heap settings and repeat test cycle until satisfactory performance is achieved.

5. Optionally, once the application is using a suitable heap size, you may wish to test different GC policies and how they interact with your application.

   Refer to 5.2.1, “Garbage collection policies” on page 37 for more information about the available GC policies and how to use a different policy.

The remainder of this chapter provides guidance on some of these steps and some other considerations for switching to IBM Technology for JVM.

4.1.1 Measuring heap usage in Classic JVM

If you do not really know how much memory your application uses, you have to obtain this information before you can make an informed decision about switching to IBM Technology for JVM. There are a couple of techniques you can use to find the memory footprint of the Classic JVM.

Dump Java Virtual Machine (DMPJVM) command

The DMPJVM command dumps information about the JVM for a specified job. It includes information about the classpath, garbage collection, and threads associated with the JVM. The output goes to a spool file for the user that executes the command as shown in Figure 4-1.

Attention: DMPJVM does hold the job briefly while it takes a snap-shot of the JVM. On rare occasions, with large JVMs, this can cause a performance problem that could require a JVM restart to recover.

The following is a sample CL command:

```
DMPJVM JOB(186947/QEJBSVR/WAS60SVR)
```

```
.....................................................................
. Garbage Collection                                                .
.....................................................................
Garbage collector parameters
   Initial size: 98304 K
   Max size: 240000000 K
Current values
   Heap size: 453768 K
   Garbage collections: 115
Additional values
   JIT heap size: 188256 K
   JVM heap size: 310524 K
   Last GC cycle time: 449 ms
.....................................................................
```

Figure 4-1 Showing total heap

Figure 4-1 shows an extract of the DMPJVM output. You have to use *Heap size* value from the output to estimate if this application fits into the IBM Technology for JVM.

Note: If the DMPJVM command does not produce any output, the command may have been terminated by your interactive job. Check the *Default wait time* of your job using the CHGJOB command. It has a default value of 30 seconds. You may want to try increasing this to 300 seconds to give the command sufficient time to complete.

For more information about the DMPJVM command, refer to the System i Information Center, V5R4 at the following link:

http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp

Using the JAVAGCINFO advanced analysis command

The JAVAGCINFO command is much more cumbersome to use compared to DMPJVM but it must not have any negative impact to the JVM. The JAVAGCINFO command requires a sign-in to Start Service Tools (STRSST). Perform the following steps to gather the heap statistics:

1. Issue STRSST at a CL command line.
2. Sign in with an appropriate ID.
3. Select **1. Start a service tool**.
4. Select **4. Display/Alter/Dump**.
5. Select **1. Display/Alter storage**.
6. Select **2. Licensed Internal Code (LIC) data**.
7. Select **14. Advanced analysis**. You may have to page/scroll down to see this option.
8. Find JAVAGCINFO in the list, type 1 in the option field to select, press Enter.
9. On the “Specify Advanced Analysis Options” just press Enter with no options.

What you see on your screen is some basic information about every Classic JVM on the system that is active. To locate the job you are interested in, look for the task name. Figure 4-2 illustrates what you will see for a job named “WAS60SVR.”

```
JVM GENERAL INFORMATION:
 VM*=EBE9DA08E4056A00 TASK=B00290000FB29000 TASKNAME=MITHREAD-WAS60SVR QEJBSVR 186947
...
JVM HEAP SIZE INFORMATION. SIZES ARE IN BYTES:
 INITIAL HEAP SIZE=100663296 THRESHOLD PER NODE = 100663296 NODES = 8
 INITIAL MAXIMUM HEAP SIZE=*NOMAX CURRENT MAXIMUM HEAP SIZE=*NOMAX
 GC HEAP MANAGER*=F85E62169E002D80 SIZE=513261568 JVM HEAP*=F85E62169E000048 SIZE=298545152
   JVM PROBLEM MODE HEAP*=C77A2BBB34000048 SIZE=20545536  JIT HEAP=C3657B7BA9000048
SIZE=193298432
```

Figure 4-2 JAVAGCINFO output

The various heap sizes (in bytes) are shown in Figure 4-2. You have to use the first value of heap (*SIZE=513261568*) from the output to estimate if this application fits into IBM Technology for JVM.

4.2 Configuring garbage collection

IBM Technology for JVM offers more options for configuring memory management component than the Classic JVM. This section describes the most important options.

4.2.1 Choosing the right policy

Chapter 5, “Tuning the garbage collector” on page 35 describes the various garbage collection policies that are available with IBM Technology for JVM. Garbage collection in general is completely different compared to the Classic JVM. Typically, GC is something that users must not have to worry about but there are a couple of things to keep in mind when you are switching to the IBM Technology for JVM.

The most important thing to remember when comparing the GC behavior of the 2 VMs is that the minimum heap size parameter means something very different to each environment. The Classic JVM has its own way of doing garbage collection and it does not allow the user to alter the behavior. The only thing that a user can influence is the frequency of GC cycles. This is done by changing the initial heap size at VM startup. For example, if you set the initial heap size to 64 MB, (**-Xms64m**) GC runs for the first time after 64 MB of space has been allocated. When a GC cycle is triggered, it frees all the space that is no longer referenced by other objects in the VM. Subsequent GC cycles run when another 64 MB of space has been allocated. Therefore, in the case of the Classic JVM, the initial heap size is really not what the name implies, it is more similar to a GC allocation threshold. This has been a point of confusion for Java programmers that are not familiar with Java on i5/OS. Users of the new IBM Technology for JVM are going to find that the initial heap size simply sets the initial heap to the value specified as its name implies. The behavior of GC is controlled by the GC policy.

Therefore now you might be wondering which GC policy is best for your application. The recommendation is that you always start with the default GC policy. The default policy, optimize for throughput (optthruput) is typically used for applications where raw throughput is more important than short GC pauses. The application is stopped each time that garbage is collected. If you prefer to test another policy, it is recommended that you read about each policy in Chapter 5, “Tuning the garbage collector” on page 35.

Tip: The recommendation is that you always start with the default GC policy. The default policy, optimize for throughput (optthruput) is typically used for applications where raw throughput is more important than short GC pauses.

4.2.2 The minimum and maximum heap size

In 4.2.1, “Choosing the right policy” on page 30, this chapter discusses how the minimum heap size (*-Xms*) is a special GC parameter for the Classic JVM. When you switch to IBM Technology for JVM the minimum heap size simply defines the minimum size of the VM. This is a significant change, therefore you must make sure that the minimum and maximum heap size are set to values that make sense for the new IBM Technology for JVM. Table 4-1 shows the default values.

Table 4-1 Default heap sizes

VM	Default value for -Xms	Default value for -Xmx
Classic JVM	16MB	*NOMAX
IBM Technology for JVM	4MB	2GB

To give an example we include Table 4-2 to show you how the WebSphere Application Server product changes these values when switching between Classic JVM and IBM Technology for JVM. For additional help choosing appropriate values for minimum and maximum heap, refer to 5.2.2, “Tuning heap size” on page 41.

Table 4-2 Default heap size values for WebSphere Application Server V6.1

VM	Default value for -Xms	Default value for -Xmx
Classic JVM	96MB	*NOMAX
IBM Technology for JVM	50MB	256MB

4.3 Finding dependencies to the Classic JVM

There must not be too many dependencies to be concerned about, but this section has a few tips that might help you have a smoother transition to the IBM Technology for JVM:

- Check for any system properties that might not be supported. For example:

  `os400.gc.heap.size.max`

  This property is an alternative to using -Xmx (setting maximum GC size) for the Classic JVM, but it does *not* have any affect on the IBM Technology for JVM. If you are using this property with the Classic JVM, it is recommended that you use the -Xmx instead. The same is true for *os400.gc.heap.size.min*, use the -Xms command line argument.

  The i5/OS Information Center has a complete list of supported properties for Classic JVM. Use the following link to view them:

  `http://publib.boulder.ibm.com/infocenter/iseries/v5r4/topic/rzaha/sysprop2.htm`

  Most of the properties found in the i5/OS Information Center are only applicable to Classic JVM. Table 4-3 shows properties that are applicable to both Classic JVM and IBM Technology for JVM.

Table 4-3 Common properties of Classic JVM and IBM Technology for JVM

Properties	Description
file.encoding	Maps the coded character set identifier (CCSID) to the corresponding ISO American Standard Code for Information Interchange (ASCII) CCSID. Also, sets the file.encoding value to the Java value that represents the ISO ASCII CCSID. See file.encoding values and iSeries CCSID for a table that shows the relationship between possible file.encoding values and the closest matching CCSID. ISO8859_1 is the default value
java.class.path	Designates the path that i5/OS uses to locate classes. Defaults to the user-specified CLASSPATH.
java.compiler	Specifies whether you use the just-in-time (JIT) compiler or not. For Classic JVM the value can be `jitc` (JIT compiler) or `jitc_de` (both the JIT compiler and direct execution). For IBM Technology for JVM the values are `NONE` or `j9jit23`. If you specify any other value, JIT compiler just ignores the setting and acts as though you specified `j9jit23`.
os400.stdio.convert	Allows control of the data conversion for stdin, stdout, and stderr in Java. Data conversion occurs by default in the Java virtual machine to convert ASCII data to or from Extended Binary Coded Decimal Interchange Code (EBCDIC). You can turn these conversions on or off with this property, which affects the current Java program.

Note: Refer to 4.2, “Configuring garbage collection” on page 30 for recommended values of maximum and minimum heap size.

- If your Java program uses adopted authority, it has to be modified. Refer to 7.3.4, “Adopted authority” on page 77 for more information.

- If your application calls native methods, they must be recompiled for teraspace as described in 7.3.1, “Teraspace storage model” on page 76.
- This might seem obvious but the best way to find a dependency is to give the IBM Technology for JVM a try. If it fails, try the following:
  a. Verify that it was working at the same JDK when it was running in Classic JVM mode. The problem can be JDK related if it was working at JDK 1.4.
  b. Look for any environment variables or system properties that can cause a problem. Use 3.3, “Basic configuration” on page 23 for reference.

4.4 WebSphere Application Server and IBM Technology for JVM

To answer the question, “Does WebSphere Application Server use IBM Technology for JVM?.” You can configure WebSphere Application Server V6.1 to use the Classic JVM or the IBM Technology for JVM. The default setting is to use the Classic JVM.

To make sure that switching WebSphere Application Server to use IBM Technology for JVM is a good idea, review the previous topics in this chapter:

- Refer to 4.1, “Fitting your application into a 32-bit JVM” on page 28. Notice, however, that the practical limit for the Java heap running WebSphere Application Server is 2.5 GB.
- Refer to 4.3, “Finding dependencies to the Classic JVM” on page 32.

If you decide to switch your WebSphere Application Server to use IBM Technology for JVM, then refer to Appendix A, “Running WebSphere Application Server with IBM Technology for JVM” on page 167. It has detailed instructions about how to make the switch.

5

Tuning the garbage collector

This chapter provides details on the operation and tuning of the garbage collector component of the IBM Technology for JVM. Garbage collection is performed differently in IBM Technology for JVM compared to Classic JVM. Four policies are available to choose from which provide different garbage collection characteristics. In some circumstances choosing a different policy might provide better performance, depending on the metrics by which application performance is measured.

IBM Technology for JVM includes an option to produce verbose output, which gives great detail regarding the operation of the garbage collector. You can analyze this information to give insight regarding whether adjusting the garbage collection policy and command-line parameter settings might lead to increased garbage collection performance.

This chapter covers the following topics:

5.1 Introduction to garbage collection

A key differentiator of Java compared to many prominent programming languages is the automatic allocation and deallocation of memory by the Java Virtual Machine (JVM). This removes the burden of explicit memory management from the programmer, reducing the scope for errors and memory leaks.

Memory management involves two distinct but related functions of the JVM:

- A *memory allocator* to allocate areas of storage for objects
- A *garbage collector* to free memory when it is no longer being used

Allocating memory is an inexpensive and speedy operation compared to reclaiming unused space. Therefore, in discussing memory management in the JVM, we focus on the more time-consuming process of reclaiming unused storage, commonly referred to as *Garbage Collection* (GC).

When a JVM initializes, it reserves an area of system memory called the *heap*. As the Java application runs and objects are created, these are stored in the heap. Because system memory is finite, the heap also has a finite size. The heap has an initial, or minimum, size and also a maximum size. The heap size can expand or contract automatically during the lifetime of the JVM as required by the garbage collector.

The garbage collector must ensure that there is enough free space in the heap to satisfy new allocation requests. It does this by *marking* live objects and *sweeping* dead objects to free memory. These are objects to which there are no references from elsewhere in the JVM, such as from other objects. You can then use this memory for new allocations.

If the heap runs out of space for some reason, it can cause the JVM to fail with an `OutOfMemoryException` message and other messages to this effect. Therefore, garbage collection is an important function of the JVM.

5.1.1 Garbage collection in IBM Technology for JVM

The Garbage Collection component has always been present in IBM JVMs. However the way in which GC is implemented in the IBM Technology for JVM shipped with i5/OS V5R4, differs compared to the Classic JVM.

In the Classic JVM, the Garbage Collector runs asynchronously (in the background) and concurrently (at the same time) to application threads. This collector is unique to i5/OS, it does not use “Stop The World” approach for collecting objects.

Tuning options in the Classic JVM are limited to setting the minimum heap size and maximum heap size. Setting the minimum heap size provides rudimentary control over the frequency of GC cycles. There is no compaction of the heap to reduce *fragmentation*. This means that the heap can grow to a larger size than actually required by the application.

Fragmentation occurs because Java objects require contiguous storage. Over the lifetime of the application, many allocation and free operations take place, each involving different size chunks of memory. This gradually results in the heap containing many small fragments of memory. These fragments might represent a significant chunk of memory in total, yet there is insufficient contiguous storage to fulfill new allocation requests. Eliminating fragmentation is therefore a key requirement of the garbage collector. GC in IBM Technology for JVM uses a compaction technique which moves all live object to the beginning of the heap. Thus, compaction aggregates all free chunks of memory in a single, continuous piece of memory.

With the introduction of the IBM Technology for JVM, GC operation and tuning methods available on i5/OS are now consistent with other platforms. The IBM Technology for JVM defines several GC *policies* which enable you to tailor the way that GC interacts with your Java application. There are four such GC policies available to choose from. These are discussed in 5.2.1, “Garbage collection policies” on page 37

There is also scope for fine-tuning other garbage collection attributes to meet your application’s requirements which are described in 5.2, “Available options” on page 37.

Heap size considerations

If your application currently uses the Classic JVM and you are considering switching to IBM Technology for JVM, the heap size requirement of your application is an important consideration. The maximum heap size with the Classic JVM is approximately 243 GB. There is practically no application that requires such a big heap. With IBM Technology for JVM, the maximum theoretical heap size is 4 GB, an inherent limitation of using a 32-bit address space. In practice, expect around 2.5 GB to 3 GB actual memory to be available for the heap due to the JVM operational requirements.

It is possible that your application running on Classic JVM uses a heap larger than 3 GB and therefore you might think that it will not run under the IBM Technology for JVM. However, bear in mind that object references are twice the size in a 64-bit JVM compared to a 32-bit JVM. Therefore, object references require more storage and result in a larger memory footprint in the Classic JVM.

Conversely, running the application under IBM Technology for JVM requires less storage for object references and therefore the memory footprint is reduced. In this way, applications that seemingly are not going to fit within the address space of the IBM Technology for JVM, can due to the contraction in size of object references and other factors.

In testing, memory footprints of applications decreased by between 20% and 50% when moved from Classic JVM to IBM Technology for JVM. Therefore, applications running under Classic JVM with up to approximately a 4 GB heap are good candidates to run under IBM Technology for JVM. If your current application requires up to 6 GB of memory, IBM Technology for JVM might or might not fit this application. You have to perform stress testing of your application running in 32-bit JVM to understand memory requirements.

Refer to 4.1.1, “Measuring heap usage in Classic JVM” on page 29 for more information about how to determine the current heap size of the application under the Classic JVM.

5.2 Available options

This section discusses the four garbage collection policies available in IBM Technology for JVM and also options for tuning the heap size and generating verbose garbage collector output.

5.2.1 Garbage collection policies

Policies are used to determine the garbage collection algorithm that the JVM uses. There are four GC policies available in IBM Technology for JVM version 5.0 and each has different characteristics. You may ask why there is so much choice. In some circumstances the different characteristics of a particular policy might lead to better application performance, depending on your performance goals.

Note: The default GC policy works well in most situations. Although changing the GC policy is straightforward, it may have a negative impact on application performance. It is recommended that you thoroughly test the new policy in a test environment before considering to use the new policy in production.

The garbage collection component in IBM Technology for JVM has a variety of functions available at its disposal, such as concurrent marking and concurrent sweeping. Behind-the-scenes, each GC policy defines which of these functions must be used in which combination to achieve a particular goal.

Table 5-1 lists the available policies and explains when you must use each one.

Table 5-1 GC policies available in IBM Technology for JVM

Policy	Command-line parameter	Description
Optimize for throughput	-Xgcpolicy:optthruput (optional)	The default policy. Typically used for applications where raw throughput is more important than short GC pauses. The application is stopped each time that garbage is collected.
Optimize for pause time	-Xgcpolicy:optavgpause	Trades high throughput for shorter GC pauses by performing some garbage collection concurrently. The application is paused for shorter periods compared to optthruput.
Generational concurrent	-Xgcpolicy:gencon	Handles short-lived objects differently than objects that are long-lived. Applications that have many short-lived objects can see shorter pause times with this policy while still producing good throughput.
Subpooling	-Xgcpolicy:subpool	Uses an algorithm similar to the default policy but employs an allocation strategy that is more suitable for multiprocessor machines. Recommended for symmetric multiprocessing (SMP) machines with 16 or more processors. Applications that have to scale on large machines can benefit from this policy.

If you want to use a policy other than the default optthruput policy, you have to use the **-Xgcpolicy** command line parameter with an appropriate argument when launching the JVM. If you do not specify the -Xgcpolicy parameter, the default policy is used.

For example, to use the Optimize for pause time (optavgpause) GC policy, you use the following command line:

```
java -Xgcpolicy:optavgpause HelloWorld
```

The following sections describe the characteristics of each policy in more detail.

Optimize for throughput (optthruput)

This is the default GC policy used in IBM Technology for JVM. It is designed to provide the best performance for the majority of applications. It does this by focusing on application throughput as the main indicator of performance.

The optthruput policy uses a mark-sweep-compact garbage collector. When a GC cycle starts, all application threads are paused. Because of this the optthruput policy is known as a “Stop The World” or STW collector.

All objects are then traced by the collector and those that are live (those objects which are still referred to by other objects or elsewhere in the JVM) are marked.

After the mark phase is complete, the collector ‘sweeps’ objects that were not marked (which are the dead objects), freeing the space. After the sweep phase, memory contains areas of free space and areas occupied by live objects.

The mark and sweep phases take place during every GC cycle. At the end of the sweep phase there may still be insufficient memory to satisfy the allocation request. This can occur if there is excessive fragmentation of the heap.

If this scenario arises, the heap is compacted. Compaction moves objects toward the beginning of the heap, aligning them so that no free space remains between objects as shown in Figure 5-1. Compaction is an expensive operation compared to the mark and sweep phases, therefore, GC avoids it unless absolutely necessary to prevent the JVM from running out of memory.

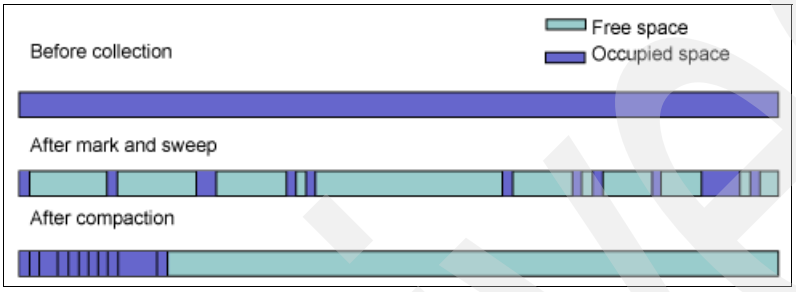


Figure 5-1 Heap layout before and after garbage collection

Optimize for pause time (optavgpause)

The default optthruput policy can lead to varying application pause times for each GC cycle because the mark and sweep work starts only after the application is paused.

For some applications, varying GC pause times can be undesirable. For instance, graphical applications which require consistent response times to user interaction might not be best suited to optthruput.

The optavgpause policy sacrifices some application throughput in order to reduce the average GC pause time. A mark-sweep-compact collection process is still used, but the application threads perform some of the mark and sweep work themselves while the application is still running as shown in Figure 5-2. One or more low-priority GC threads also run in the background to perform concurrent marking while the application is idle.

Stop-The-World collection is still used but does not take as long as optthruput because some of the work has already been done. The throughput penalty when using optavgpause is around 5% but this varies by application.

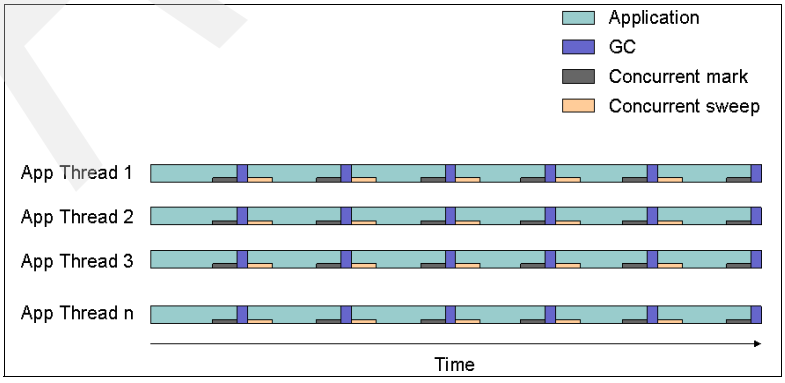


Figure 5-2 Distribution of CPU time in the optavgpause GC policy

Generational concurrent (gencon)

The generational concurrent policy exploits the so-called *weak generational hypothesis* that has emerged, which asserts that “most objects die young”. The gencon policy combines concurrent mark and generational GC to minimize GC pauses. Applications that create a lot of short-lived objects (such as transaction-based applications) might be suited to the gencon policy.

Generational GC separates the heap into two areas referred to as *nursery space* and *tenured space*. Allocations are initially in the nursery space. If an object survives enough garbage collections, it is promoted to the tenured space, and the object is said to have *tenured*. The nursery and tenured areas are collected differently for performance.

The nursery space is further split into an *allocate space* and a *survivor space* as shown in Figure 5-3. Objects are allocated to the allocate space. When the allocate space is full, a GC process called *scavenge* is triggered. The scavenge copies live objects into either the survivor space, or tenured space if they are old enough. Age is measured in terms of how many garbage collections the object has survived. The JVM determines dynamically how old an object must be to tenure, however the maximum is 14 GC cycles. Live objects are copied to the survivor space and aligned in such a way so that it can avoid fragmentation.

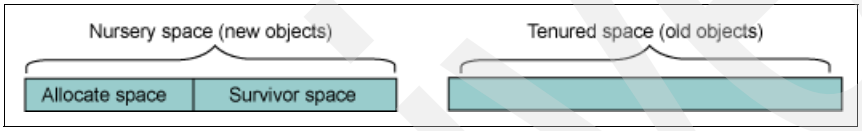


Figure 5-3 The heap is divided into nursery and tenured spaces in the gencon GC policy

After live objects have been copied into the survivor space, the allocate space contains only dead objects. The allocate space and survivor space then swap or *flip* roles, this means that the allocate space becomes the survivor space for the next scavenge and the survivor space becomes the allocate space. On the next scavenge, live objects overwrite dead objects from the previous scavenge for efficiency. The allocate and survivor spaces then flip roles again. The scavenge process runs on the nursery area of the heap only, therefore, the cost of collecting the entire heap on each GC cycle is not incurred. See Figure 5-4.

Objects in the tenured space typically live longer than those in the nursery, and therefore the tenured space is deliberately collected less often than the nursery. The tenured space is marked concurrently using a similar approach to the optavgpause policy, hence the policy is called generational concurrent (gencon). However, concurrent sweep is not used, therefore, when concurrent marking of the tenured space is completed, the application is paused for the global collection to be completed. This global collection collects garbage in both the nursery and tenured spaces.

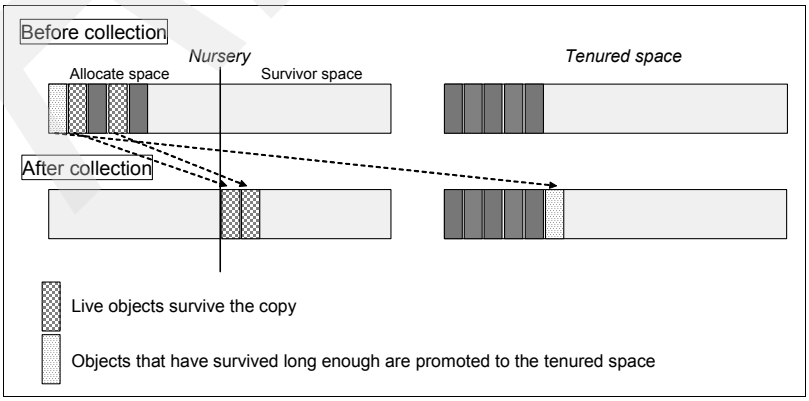


Figure 5-4 Example of heap layout before and after garbage collection using gencon policy

Subpooling (subpool)

The subpool policy is available on IBM eServer™ System i, IBM eServer System p™, and IBM eServer System z™. It is designed for use on multiprocessor machines (typically 16 processors or more). When scalability of your application on large machines is important, consider using subpooling.

All GC policies in IBM Technology for JVM employ a *free list*. The free list is a data structure that keeps track of which areas of the heap are available for allocations. The free list also contains information about the size of each area of free space. Typically a single free list is maintained for the entire heap. Much of the list might have to be searched before a suitable chunk of free space is found.

Subpooling makes use of multiple free lists referred to as *pools*. A pool contains the location of chunks of free space of a specific size. Each pool is associated with a different size as shown in Figure 5-5. The pools are ordered by size, therefore the memory allocator only has to go to the first element of the right pool to allocate an object. This results in faster object allocation compared to GC policies with a single free list, because the only search required is to locate the correct size pool to allocate from.

Subpooling also makes use of per-processor “mini heaps” which further improve performance on multiprocessor systems. These are managed automatically by the JVM, therefore no mechanism is provided for adjustment.

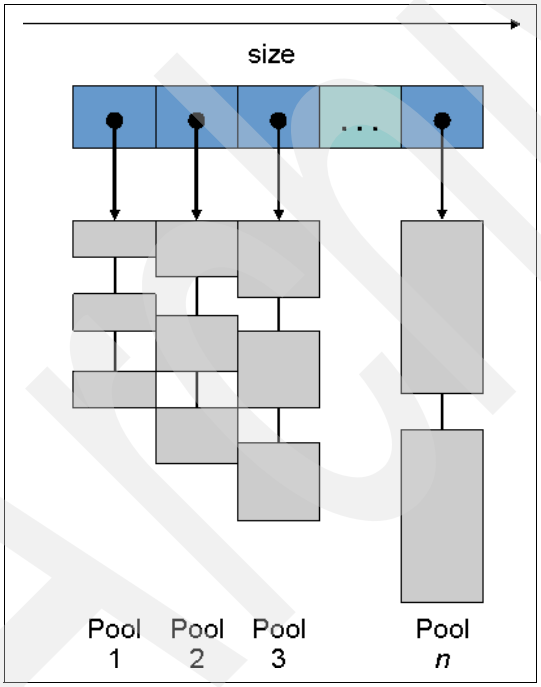


Figure 5-5 Subpooling employs multiple pools each with a list of free chunks of a specific size

5.2.2 Tuning heap size

The size of the heap available to an application can influence performance dramatically. Too small a heap, and the garbage collector has to run frequently, reducing application throughput and increasing response times. If the heap is much larger than required by the application however, garbage collection takes place rarely and takes longer because the whole heap has to be scanned for live objects. Therefore, it is important to achieve a balance. A good rule of thumb is to have a heap 1.5 times the actual size required.

As mentioned in “Heap size considerations” on page 37, applications use a smaller heap under IBM Technology for JVM compared to Classic JVM. The heap minimum and maximum size settings your application might have used with the Classic JVM will not necessarily be valid with IBM Technology for JVM. For example, an unlimited maximum heap size is not supported in IBM Technology for JVM. If you do not specify a maximum, the default of 2 GB is used. For further information, refer to 4.2, “Configuring garbage collection” on page 30.

Default heap size settings in IBM Technology for JVM

With the IBM Technology for JVM you can set both the minimum and maximum size of the heap, or you can use the default settings. On i5/OS the default minimum heap size is 4 MB and the default maximum size is 2 GB. The minimum heap size is the smallest size the heap is allowed to reach through heap contraction. The heap is initially the same size as the minimum size. The maximum heap size is the largest size the heap is allowed to reach through heap expansion. It is possible to set the minimum and maximum heap sizes to the same value. This will have the effect that the heap neither expands nor contracts in size during the life of the JVM.

To override the default heap size settings, you use the -Xms and -Xmx command-line parameters for minimum and maximum respectively. Default settings are used if these parameters are not specified.

For example, if you want to change a maximum heap size to 256 MB and use the default value for a minimum heap size, then you use Xmx parameter:

```
java -Xmx256m HelloWorld
```

To specify a minimum heap size of 32 MB and use the default maximum heap size:

```
java -Xms32m HelloWorld
```

To specify a minimum heap size of 32 MB and maximum heap size of 256 MB:

```
java -Xms32m -Xmx256m HelloWorld
```

The IBM Technology for JVM now supports use of the G or g argument to specify the heap size settings in gigabytes rather than megabytes as a convenience:

```
java -Xmx1g HelloWorld
```

This sets the maximum heap size to 1 GB. Only whole numbers of gigabytes are allowed, therefore, specifying -Xmx1.5g returns an error.

If you try to specify too large a maximum heap, the java command returns the following error:

```
JVMJ9GC028E Option too large
```

Determining appropriate heap size settings

If you are switching to IBM Technology for JVM from Classic JVM, it is recommended that you run the application with the default heap size settings initially and verbose GC output enabled. Refer to 5.2.3, “Verbose GC output” on page 44. You can then use the following approach to adjust the minimum and maximum heap size settings appropriately. This approach is also valid even if the application was not previously run under Classic JVM.

The application must be run up to steady state with no load. Analyze the verbose GC output at this stage. The current heap size, reported in the most recent entry in the verbose GC output, gives you a rough indication of the initial size of the heap you must use (defined by the -Xms command-line parameter).

If the value reported is much larger than the default value of 4 MB, consider using a slightly lower than reported value for the minimum heap size setting. For example, if the reported heap value is 192 MB, then set the minimum heap size of your JVM to 170 MB. This gives more efficient compaction early on compared to using a large value.

Next, you must run the application under stress to determine an appropriate value for the maximum heap size. When the application is processing the highest workload, check the verbose GC output for the current heap size. Use a slightly larger value to set the maximum heap size (-Xmx command-line parameter). For example, if JVM reports 650 MB as a heap size under the highest workload, set the maximum heap size for your JVM to 680 MB. This gives your JVM a small cushion in the production environment.

Heap expansion and contraction

Unless you fix the size of the heap at JVM startup by setting the minimum and maximum heap sizes to the same value, the heap expands and sometimes contracts automatically as required by the garbage collector to maintain efficient operation. The size of the heap, however, never contracts below minimum size, and never expands beyond maximum size. The current “top of the heap” is represented by `heaptop` in Figure 5-6.

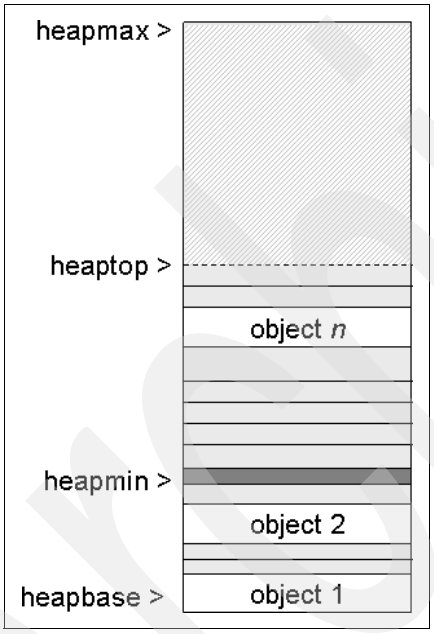


Figure 5-6 Layout of the heap

For fixed-size heaps there is no concept of `heaptop`. For non-fixed heaps, `heaptop` may move toward `heapmin` or `heapmax`, depending how much free space is in the heap after each collection.

The garbage collector by default tries to maintain a minimum of 30% free space in the heap for new allocations. If less than 30% free space is available after garbage has been collected in the current GC cycle, the collector might expand the heap by the amount required to reach 30% free space. You can override the minimum free space the garbage collector tries to maintain by specifying the -Xminf command-line parameter. Specify the size as a decimal in the range 0-1.

For example to maintain 20% minimum free space in the heap, specify the following:

```
java -Xminf0.2 HelloWorld
```

If there is a lot of free space in the heap after garbage has been collected, the garbage collector may contract the heap. This can improve performance because a smaller heap can be scanned for live objects faster. By default, if more than 60% of the heap is free space, the garbage collector contracts the heap so that a maximum of 60% free space exists. You can override the maximum free space by specifying the -Xmaxf command-line parameter. Specify the size as a decimal in the range 0-1.

For example to maintain a maximum of 50% free space in the heap, specify the following:

```
java -XmaxfO.5 HelloWorld
```

Note: As with other GC related settings, the default -Xminf and -Xmaxf settings work well, therefore, most users do not have to adjust them.

5.2.3 Verbose GC output

IBM Technology for JVM includes an option to produce verbose GC output to give a detailed insight into the operation of the garbage collector. It is possible to monitor the size of the heap through analyzing verbose GC output. This enables you to determine the smallest suitable heap size for your application and also the amount of room required for heap expansion. Analyzing verbose GC output can show whether using a different GC policy has a positive or negative impact on the GC characteristics of your application.

Verbose GC output produced by IBM Technology for JVM is now formatted as Extensible Markup Language (XML). This allows tools to be written to analyze verbose GC output automatically.

You can collect verbose GC output for review if you want to analyze the GC characteristics and performance of an application. You can use the analysis of the information in verbose GC output to support a decision to move to a different GC policy or to tune GC options such as the heap size. Verbose GC output can also help you identify possible causes of poor GC performance.

Verbose GC output is enabled by setting one of two command-line options when launching the JVM, either **-verbose:gc** (or **-verbosegc**) or **-Xverbosegclog:filename. -verbose:gc** writes its output to the standard error stream, while **-Xverbosegclog** writes its output to the file `filename`.

For example, to enable verbose GC output and capture the output in a file GClog.txt stored in the Integrated File System (IFS), use the following:

```
java -Xverbosegclog:/home/rumsbya/GClog.txt HelloWorld
```

Tip: To analyze the GC output with the tools, use the *verbosegclog* option. Tools do not read the standard error stream correctly because it might contain some other data, for example, exception stacks.

Interpreting verbose GC output

This section explores the following extract from a verbose GC output file shown in Figure 5-7 to identify the key elements which you can use to determine how GC is performing. For this example we ran WebSphere Application Server V6.1.0.1 Base with an artificially small heap size for demonstration purposes. We also started the Trade 6 performance benchmark sample application with 500 virtual users to generate load on the server. For more information about the Trade 6 application, refer to “Trade 6 application” on page 133.

The results are not typical, but nevertheless they show how much useful information can be gained from reviewing verbose GC output.

```
...
<af type="tenured" id="36" timestamp="Mon Oct 09 13:58:15 2006" intervalms="382.636">
  <minimum requested_bytes="972016" />
  <time exclusiveaccessms="0.117" />
  <tenured freebytes="14877752" totalbytes="130819584" percent="11" >
    <soa freebytes="14877752" totalbytes="130819584" percent="11" />
    <loa freebytes="0" totalbytes="0" percent="0" />
  </tenured>
  <gc type="global" id="36" totalid="36" intervalms="85383.105">
    <compaction movecount="885558" movebytes="53290192" reason="compact to meet allocation" />
    <classloadersunloaded count="58" timetakenms="11.881" />
    <refs_cleared soft="7" weak="429" phantom="152" />
    <finalization objectsqueued="336" />
    <timesms mark="233.177" sweep="8.977" compact="527.274" total="781.744" />
    <tenured freebytes="77525984" totalbytes="130819584" percent="59" >
      <soa freebytes="76217824" totalbytes="129511424" percent="58" />
      <loa freebytes="1308160" totalbytes="1308160" percent="100" />
    </tenured>
  </gc>
  <tenured freebytes="76553968" totalbytes="130819584" percent="58" >
    <soa freebytes="75245808" totalbytes="129511424" percent="58" />
    <loa freebytes="1308160" totalbytes="1308160" percent="100" />
  </tenured>
  <time totalms="782.403" />
</af>
...
```

Figure 5-7 Extract from verbose GC log for one collection using optthruput policy

The top-level element <af> indicates this GC cycle occurred due to an allocation failure. Other possible values you might see are <sys> for collections forced by System.gc() calls and <con> for concurrent collections.

Tip: <sys> elements in verbose GC output indicate the application is making calls to System.gc(). You are strongly discouraged to use System.gc() calls to try to influence garbage collection because this can severely degrade performance. Remove System.gc() calls from the application or use the **-Xdisableexplicitgc** command-line option when launching the JVM so that System.gc() calls have no effect.

Note the intervalms attribute in the first line:

```
<af type="tenured" id="36" timestamp="Mon Oct 09 13:58:15 2006"
intervalms="382.636">
```

This shows that only 382.636 milliseconds have passed since the last allocation failure. This can be an indication that the heap is filling up too quickly. Consider increasing the maximum heap size.

The next line shows the size of the allocation request that caused the failure, in this case nearly 950 KB:

```
<minimum requested_bytes="972016" />
```

Notice that there are three <tenured> elements for each garbage collection. These describe the state of the heap at different stages of the GC cycle:

- The first represents the heap prior to GC, right after the allocation failure occurred.
- The second represents the heap after collection has occurred.
- The third shows heap occupancy after the request which originally caused the allocation failure has been satisfied.

> **Note:** It is recommended that you use the value of *totalbytes* from the first <tenured> elements when you estimate the maximum heap size setting of your JVM.

The verbose GC output shows the effect of fragmentation on the heap:

```
<tenured freebytes="14877752" totalbytes="130819584" percent="11" >
```

Although there is over 14 MB total free space in the heap, there is not a large enough contiguous chunk to allocate the requested 950 KB. Therefore, compaction is triggered to move objects toward the beginning of the heap. On this occasion over 880,000 objects are moved:

```
<compaction movecount="885558" movebytes="53290192" reason="compact to meet
allocation" />
```

From the verbose GC output you can see the high cost of compaction compared to the mark and sweep phases of the GC cycle. This line shows the time in milliseconds taken for each phase of the GC cycle:

```
<timesms mark="233.177" sweep="8.977" compact="527.274" total="781.744" />
```

On this occasion, compaction took more than twice as long than the mark phase and nearly 60 times longer than sweeping dead objects. The whole GC cycle took nearly 0.8 seconds. This emphasizes the importance of avoiding compaction.

One approach is to accept that compaction is required at some stage in an application lifetime, and to minimize the impact by setting a small minimum heap size. The heap requires compaction early on, but because the heap is small it does not take as long as when the application has reached steady state and expanded toward the maximum size. Early compaction has the effect that most long-lived objects are at the beginning of the heap by the time the heap starts expanding. By comparison, a large initial heap takes longer to fill and is more fragmented by the time the first garbage collection occurs. Long-lived objects will be scattered throughout the heap and costly to relocate to the beginning of the heap.

Another approach is to use the generational concurrent (gencon) GC policy, particularly if your application creates a lot of short-lived objects. Because gencon uses a copying collector, efficient compaction of the nursery occurs as a side-effect of each scavenge, versus expensive compaction of the entire heap in other policies. However, we must point out that the nursery collection happens very frequently.

5.3 Choosing the right policy

It is important to consider garbage collection in the context of overall application performance. For most enterprise applications, the JVM is just one component in a system which may also include databases, network infrastructure, Lightweight Directory Access Protocol (LDAP) directories, Web servers, application servers, and other components. Each of these can contribute positively or negatively to application performance. The application itself can be a bottleneck if poorly written.

Therefore, you must consider the possible performance gains carefully before investing effort in tuning garbage collection or adjusting the GC policy. For instance, if your Web server is unable to keep up with the number of incoming requests, then there is little to be gained by tuning the garbage collector. On the other hand, if you have a mature and proven application environment that has been well-tested, you might want to consider tuning garbage collection as part of a wider tuning strategy for your application.

Before you consider changing the garbage collection policy, you have to determine which performance characteristics are important for your application. *Throughput* and *response time* are two measurements of an application's performance that you must consider. Throughput measures how much data is processed by the application in a given time period, usually expressed as operations or transactions per second. Response time measures how long an application takes to complete processing of a request, starting from when the request is received by the application.

Different applications place different importance on each of these performance measurements. An interactive GUI-based application, for example, typically requires to have consistently low response times. If the response time varies wildly, users are not satisfied with the performance. A batch payroll application on the other hand, has to process a lot of data in a given time period (high throughput), while the time taken to respond to each individual request is of little consequence.

The default garbage collection policy is optimized to provide high application throughput while providing GC pauses short enough for the majority of applications. It is recommended that you run the application using the default GC policy initially. If this gives satisfactory performance, then there is no requirement to try other GC policies, although doing so is straightforward. Just specify the relevant -Xgcpolicy:policy command-line parameter when launching the JVM.

Note: Before considering changing the GC policy, you must be using a heap size appropriate to your application's requirements. For example, if the maximum heap size is set too small, then performance suffers regardless of which GC policy you choose.

If the default GC policy does not give satisfactory performance, analyze verbose GC output as described in "Interpreting verbose GC output" on page 44, looking for clues to find the particular aspect of GC behavior which is causing concern.

Tip: There is an excellent article available from IBM which presents a quantitative approach to choosing a GC policy using a case study. The article is available from IBM developerWorks® at:

`http://www-128.ibm.com/developerworks/java/library/j-ibmjava3/`

For instance, excessive fragmentation of the heap might be occurring if the application is creating many short-lived objects. As a result, you will see in the verbose GC output that compaction is happening too often, increasing the GC pause time. Subsequently the application response time may increase, although the extent to which this affects the response time depends on the workload. The effect is more noticeable in applications that are lightly-loaded compared to heavily-loaded applications in which GC pause time makes up only a small fraction of a response time which also includes more significant factors such as network latency. Switching to the generational concurrent GC policy reduces fragmentation and therefore reduces pause times.

For interactive applications where the response time is varying unpredictably, the verbose GC output might show that there are long pauses for GC. In this case consider using the following policy:

```
optavgpause
```

If you have a multiprocessor server with 16 or more processors (such as a 16-way IBM System i5™ 570 or IBM System i5 595) consider using subpooling for faster object allocation. Per-processor “mini heaps” used automatically in this policy might further improve performance. In addition to IBM System i5, subpooling is also available on IBM System p5™ and IBM System z9™ platforms.

6

Optimizing performance with shared classes

This chapter discusses the new shared classes feature available in IBM Technology for JVM. Shared classes are loaded once and shared by multiple JVMs that are running on the server. This can improve JVM startup times if the required classes have been loaded previously by another JVM.

This chapter presents an overview of class loading, then explains how shared classes fit into the class loading model. The details of the shared classes featured in IBM Technology for JVM are then explained.

The effect of shared classes on application performance is then discussed. This chapter gives an overview of how you can integrate shared classes support into your own custom-written classloaders and mention security considerations for using shared classes.

This chapter is divided into the following sections:

6.1 Introduction to shared classes

If you have never used shared classes before, you must read this section. It helps you to comprehend information presented in the rest of this chapter.

6.1.1 Classes and classloaders

Unlike languages such as C which are linked statically at compile time, Java is a dynamically-linked language. This means that the classes required by a Java application are loaded into the runtime environment as required during application execution.

You must load classes into memory before you can create any instances of those classes, or any static data referenced by other classes. You can load classes from several sources, such as from a .class, zip, or Java archive (JAR) file present in the local filesystem, or from a remote server.

Categories of classloader

Classes are loaded by a classloader. There are several classloaders supplied with IBM Technology for JVM, which are used for different class-loading purposes:

- Bootstrap classloader

  Loads classes from the boot class path (typically classes under jre/lib, such as classes stored in rt.jar). These classes constitute the core of the runtime environment required by all applications (such as Object or RuntimeException). The bootstrap class loader is built-in to the JVM.

- Extension classloader

  Loads classes from the installed extensions located on the extension class path. For example, the IBM JCE provider classes which are included as a value-add feature of IBM JVMs. In IBM Technology for JVM on i5/OS, such extensions are found in the following directory by default:

  `/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/ext`

- System classloader (also known as Application classloader)

  Loads classes from the general class path (java.class.path system property) such as the classes that make up an application.

When a classloader is asked to load a class, it searches for the class in the following order:

- Classloader cache: if this classloader has previously loaded the class, it reads the class from its cache.
- Parent classloader
- Shared class cache: if shared classes are enabled. Refer to 6.2, “Shared classes in IBM Technology for JVM” on page 53.
- Filesystem

From this you can see a *delegation model* is used to load classes, as shown in Figure 6-1. The particular delegation model used in IBM Technology for JVM is referred to as *parent first* delegation. This means that if the current classloader has not already loaded the class, it asks its parent classloader if it has already loaded the class. If the parent has not loaded the class already, it delegates to *its* parent, and so on up to the bootstrap classloader.

If the class is not already loaded by any classloader, the classloader which was initially asked to load the class, does so. The class is loaded from the filesystem, or possibly from the shared class cache, if shared classes are enabled.

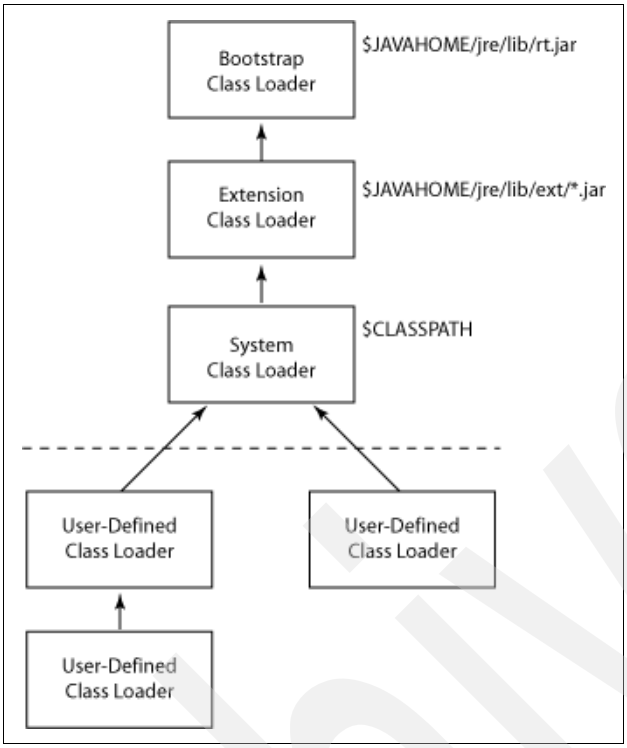


Figure 6-1 Classloader delegation model

The usage of these different classloaders is largely transparent to the programmer. The programmer must be aware of the order in which the runtime environment searches for classes, to ensure no name clashes occur. In practice, it is rare that any conflicts will occur, therefore, exactly how application classes get loaded is usually of little concern unless you are using a user-defined classloader.

User-defined classloaders

It is possible to write your own classloader in Java. This can be useful to extend class-loading behavior and capabilities. For example your application might have to:

- Find and load a class from a unique location in the file system.
- Load a different version of the class that has been loaded by another classloader. You can load the same class multiple times as long as each version of the class is loaded by a different classloader.
- Generate a class on-the-fly depending on the state of your application, or modify the bytecode to add tracing.

Custom classloaders are referred to as *user-defined* classloaders. User-defined classloaders are created by extending the abstract class java.lang.ClassLoader or one of its subclasses. This class provides implementations of various useful methods such as findClass(), loadClass(), and defineClass(), which user-defined classloaders can override for enhanced functionality. The system classloader is an example of a user-defined classloader (although it is shipped with the JVM). It is implemented as java.net.URLClassLoader. WebSphere Application Server ships with several user-defined classloaders such as a WebSphere extensions classloader and application module classloader.

Classloading and application startup

The entry point of a Java application is the `public static void main (String args[])` method, which is usually defined in the main application class. As part of loading this initial application class, there is a core set of classes that must also be loaded. The class Object, for example, as the superclass of all other objects must be loaded before any application classes. In fact, regardless of the JVM, there are hundreds of core classes that must be loaded whenever any application starts.

In traditional classloader models, these core classes are loaded into the private address space of the JVM. Therefore, if multiple JVMs are running, no JVM can see what classes other JVMs might already have loaded. Loading multiple copies of the same core classes clearly leads to a larger memory footprint for each JVM in this scenario. This aspect of the Java runtime environment is the driving force behind research into technologies such as shared classes.

6.1.2 Overview of shared classes

You can see that in the traditional class loader model, each JVM running on the system loads the classes it requires from either the local filesystem or remotely, regardless of whether another JVM has already loaded the same class into its private address space.

As mentioned in “Classloading and application startup” on page 52, hundreds of core classes have to be loaded by the JVM for any application. On a server running multiple JVMs, this leads to the same classes being loaded multiple times into each JVM. Class loading takes time and also memory in which to store the loaded classes, therefore it is clearly inefficient to load multiple copies of the same class into different areas of memory.

When enabled, shared classes remove this issue by maintaining a single copy of each class loaded by the JVMs running on the server, which is shared between all JVMs that require it. The classes are stored in shared memory which is accessible by all the JVMs running on the server. When a classloader has to load a class which was not previously loaded by a parent classloader, it looks in the shared cache in case another JVM has already loaded the required class. Only if the class is not in the cache will the classloader search the filesystem.

Because there is only one copy of a class stored in memory, sharing between JVMs reduces the memory footprint of each JVM. Retrieving a copy of the class from memory is clearly faster than loading the class from a .class, zip, or JAR file stored on disk or remotely.

Another benefit of using shared classes is that the class cache remains on the system until next initial program load (IPL). Therefore, even if you run only one JVM at a time but use frequent restarts of a JVM, you still benefit from using the shared classes feature.

6.1.3 History of shared classes

Shared classes as a concept has been around for some time, with forms of class sharing appearing in some IBM JVMs since Java 1.3. The IBM Persistent Reusable JVM on IBM z/OS® was one of the first available JVMs from IBM which included a form of class sharing, although it required JVMs to be configured in a master/slave scenario. The Classic JVM on i5/OS employs a Java cache (implemented as a shared JAR file) available to user-defined classloaders, allowing reuse of JVAPGM objects and class bytecodes.

Version 5.0 of IBM Technology for JVM offers, for the first time across the major platforms, a completely transparent and dynamic means of sharing all loaded classes (with the exception of Sun Solaris™ and the HP hybrids). Furthermore, no restrictions are placed on JVMs that are sharing the class data.

As explained later, however, in 6.6, “Bytecode modification” on page 68, care must be exercised if you are using runtime bytecode modification of classes.

6.1.4 Shared classes and user-defined classloaders

In version 5.0 of IBM Technology for JVM, classes loaded by user-defined classloaders can be shared. The class java.net.URLClassLoader has been enhanced to support shared classes. Therefore, if a user-defined classloader extends this class, it will have shared classes support built-in by default due to inheritance.

IBM has provided a Shared Classes Helper application programming interface (API) so that support for shared classes can be added to user-defined classloaders which do not inherit from java.net.URLClassLoader. The helper API is described in more detail in 6.4, “Shared Classes Helper API” on page 63.

6.2 Shared classes in IBM Technology for JVM

This section discusses in more detail the implementation and usage of the shared classes feature that you can enable in IBM Technology for JVM.

6.2.1 The class cache

In IBM Technology for JVM, shared classes are facilitated by one or more caches stored in shared memory. There can be multiple caches on the server, but each JVM can connect to only a single cache in its lifetime. When shared classes are enabled, classloaders store classes in a cache for use by other JVMs and find classes in the cache. The cache supports multiple JVMs reading/writing concurrently.

In addition to a cache, the first JVM to start with shared classes enabled creates the/tmp/javasharedresources directory. This directory holds information about all the caches on the server. Because the cache is stored in shared memory, a cache can persist only until the next IPL, at which time it is lost. One or all caches can be explicitly destroyed at any time if required, without the necessity to IPL.

A cache is created with a fixed size which cannot change after the cache is created. After the cache is full, classes can still be loaded from the cache, but no more classes can be stored. The default cache size on i5/OS is 16 MB. This can be overridden at runtime using the -Xscmx command-line parameter. Refer to “Setting class cache size” on page 59.

The cache only contains immutable (read-only) data from classes, such as fields declared as *final static*. Such data is stored in the cache as an internal structure called a ROMClass. Data which could change is stored separately in a structure called a RAMClass. They are stored in the JVMs private process memory and point to the relevant ROMClass. Multiple RAMClasses can point to the same ROMClass because the ROMClass contains read-only data. This is the fundamental principle on which class sharing is based. The distinction between ROMClass and RAMClass is managed by the JVM and is transparent to classloaders.

Metadata is stored in the cache along with the ROMClass consisting of versioning information and information about where each class was loaded from. This is used to make sure the correct class is loaded from the cache in cases where the on-disk representation of the class changes during the cache lifetime or if the same class exists on a different class path.

When shared classes are enabled, the delegation model outlined in “Categories of classloader” on page 50 still applies. This means that parent classloaders up to the bootstrap classloader are asked in turn for the class in case they have already loaded it. If no parent classloader has loaded the class previously, rather than going straight to the filesystem to locate the class, the shared classes cache is searched. Only if the class is not found in the cache will the filesystem be searched. When the class is found, it is added to the class cache, if the cache is not full.

Shared classes scope

All classes loaded by the runtime environment, both system and application, can be shared. The bootstrap, extension, and system classloaders have been modified to take advantage of the shared classes feature when it is enabled. Custom classloaders have to either extend java.net.URLClassLoader or use the Shared Classes Helper API to integrate shared classes features.

WebSphere Application Server and shared classes

Version 6.1 of WebSphere Application Server uses the shared classes feature automatically and transparently to improve performance.

6.2.2 Deploying shared classes

The shared classes feature is disabled by default in IBM Technology for JVM. Shared classes are enabled and managed through use of the -Xshareclasses command-line parameter and its suboptions.

Important: It is recommended that you apply program temporary fix (PTF) SI25920. This fixes an issue whereby a class name had to be specified for -Xshareclasses suboptions to work, even for the utility options.

Enabling shared classes

Shared classes are enabled implicitly by creating at least one class cache for JVMs to store shared classes in. To create a new class cache, the `name=` suboption is used when starting an application. If a cache of the same name already exists, the JVM connects to the existing cache. Figure 6-2 is an example which creates two caches.

```
                                QSH Command Entry

 > java -Xshareclasses:name=cache1 HelloWorld
   Hello world!
   $
 > java -Xshareclasses:name=cache2 HelloWorld
   Hello world!
   $


===> ____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
 F3=Exit   F6=Print F9=Retrieve F12=Disconnect
 F13=Clear F17=Top  F18=Bottom  F21=CL command entry
```

Figure 6-2 Creating two class caches

Listing all class caches

To show the caches that exist on the server, use the listAllCaches suboption as shown in Figure 6-3.

```
                                  QSH Command Entry

> java -Xshareclasses:listAllCaches
  Shared Cache            OS shmid        in use          Last detach time
  cache2                  5833            0               Wed Sep 27 16:05:23 2006

  cache1                  5831            0               Wed Sep 27 16:05:12 2006

  Unable to create Java Virtual Machine.
  $


===> ____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
 F3=Exit   F6=Print F9=Retrieve F12=Disconnect
 F13=Clear F17=Top  F18=Bottom  F21=CL command entry
```

Figure 6-3 The listAllCaches suboption lists all the class caches present on the system

Note: Some of the -Xshareclasses suboptions are *utility* options. Although they use the JVM launcher (the `java` command), by design a JVM is not started. Consequently, an "`Unable to create Java Virtual Machine`" message is always output by the launcher after a utility has run, as shown in Figure 6-3. This message is *not* an error.

Viewing summary class cache statistics

To view summary statistics for a named cache, use the printStats suboption as shown in Figure 6-4.

```
                                QSH Command Entry

 > java -Xshareclasses:name=cache1,printStats

   Current statistics for cache "cache1":


   base address       = 0xE0000058
   end address        = 0xE0FFFFF8
   allocation pointer = 0xE010F4A0

   cache size         = 16777128
   free bytes         = 15652864
   ROMClass bytes     = 1111112
   Metadata bytes     = 13152
   Metadata % used    = 1%

   # ROMClasses       = 278
   # Classpaths       = 2
   # URLs             = 0
   # Tokens           = 0
   # Stale classes    = 0
   % Stale classes    = 0%

   Cache is 6% full

   Unable to create Java Virtual Machine.
   $

===> ____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
 F3=Exit   F6=Print F9=Retrieve F12=Disconnect
 F13=Clear F17=Top  F18=Bottom  F21=CL command entry
```

Figure 6-4 Displaying summary cache statistics using printStats suboption

Viewing class cache contents

The printAllStats suboption lists the entire contents of the cache, and also displays the summary statistical information returned by the printStats suboption. The output is listed in the chronological order by which classes are added to the cache, therefore you can use them as an “audit trail”. Each JVM that connects to the cache receives a unique ID which is shown preceding each entry in Figure 6-5.

```
                                QSH Command Entry

> java -Xshareclasses:name=cache1,printAllStats

  Current statistics for cache "cache1":

  1: 0xE0FFF8B4 CLASSPATH
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/vm.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/core.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/charsets.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/graphics.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/security.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/ibmpkcs.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/ibmorb.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/ibmcfw.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/ibmorbapi.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/ibmjcefw.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/ibmjgssprovider.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/ibmjsseprovider2.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/ibmjaaslm.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/ibmcertpathprovider.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/server.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/xml.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/IBMmisc.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/lib/IBMi5OSJSSE.jar
  1: 0xE0FFF87C ROMCLASS: java/lang/Object at 0xE0000058.
          Index 0 in classpath 0xE0FFF8B4
  1: 0xE0FFF854 ROMCLASS: java/lang/J9VMInternals at 0xE00006E0.
          Index 0 in classpath 0xE0FFF8B4
  1: 0xE0FFF82C ROMCLASS: java/lang/Class at 0xE00020A8.
          Index 0 in classpath 0xE0FFF8B4
       ...
```

Figure 6-5 Displaying class cache contents using PrintAllStats suboption

Interpreting printAllStats output

The output shown in Figure 6-5 indicates that JVM 1 (the first JVM to use this cache) caused a classpath to be stored at address 0xE0FFF8B4 in the cache. The classpath contains 18 entries, which are listed. If the classpath was stored using a given modification context or partition (refer to 6.6, “Bytecode modification” on page 68) this information is also displayed.

Taking a closer look at the meaning of the address information given for each ROMClass in the cache, in this case java.lang.Object:

```
1: 0xE0FFF87C ROMCLASS: java/lang/Object at 0xE0000058.
           Index 0 in classpath 0xE0FFF8B4
```

The first address, 0xE0FFF87C, is the address where the metadata for this ROMClass is stored. The second address 0xE0000058 is the address where the ROMClass itself is located in the cache. The next line gives the location of the classpath from which this class was loaded (0xE0FFF8B4) and the index of this class in the classpath, which is 0 in this case. The classpath is *zero-indexed*, therefore, 0 corresponds to the first entry in the classpath in Figure 6-5. This means that java.lang.Object was loaded from the following as expected:

```
/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/vm.jar
```

Deleting the class cache

It is possible to delete one or all caches at any time using the destroy and destroyAll utility suboptions respectively, as shown in Figure 6-6.

```
                                QSH Command Entry

> java -Xshareclasses:name=cache1,destroy
  JVMSHRC010I Shared Cache "cache1" is destroyed
  Unable to create Java Virtual Machine.
  $
> java -Xshareclasses:listAllCaches
  Shared Cache            OS shmid       in use          Last detach time
  cache2                  5833           0               Wed Sep 27 16:05:23 2006

  Unable to create Java Virtual Machine.
  $

===> ____________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
 F3=Exit   F6=Print F9=Retrieve F12=Disconnect
 F13=Clear F17=Top  F18=Bottom  F21=CL command entry
```

Figure 6-6 Deleting a class cache

There might be several reasons why you want to delete a class cache. One example is if you want to recreate a class cache with a different size. Another example is if you do a lot of testing with the different versions of your applications, your class cache may contain many classes that you do not require anymore. In order to clean the cache, you delete it. Next time you start your applications, the JVM creates a clean class cache.

Important: Only the user profile under which a cache was created, may delete the cache. This is true even if the groupAccess suboption was specified when the cache was created. Refer to 6.5.1, “Operating system security” on page 67 for more information about the groupAccess suboption.

Setting class cache size

The cache size can be set using the -Xscmx command-line parameter, as shown in Figure 6-7.

Notice the use of the verbose suboption to show that the class cache has become full when using an artificially small cache size of 10 KB.

```
                                QSH Command Entry

 > java -Xscmx10k -Xshareclasses:name=cache1,verbose HelloWorld
   ^-Xshareclasses verbose output enabled]
   JVMSHRC159I Successfully opened shared class cache "cache1"
   JVMSHRC166I Attached to cache "cache1", size=10200 bytes
   JVMSHRC096I WARNING: Shared Cache "cache1" is full. Use -Xscmx to set cache size.
   Hello world!
   JVMSHRC168I Total shared class bytes read=1672. Total bytes stored=0
   $

===> ____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
 F3=Exit   F6=Print F9=Retrieve F12=Disconnect
 F13=Clear F17=Top  F18=Bottom  F21=CL command entry
```

Figure 6-7 Setting class cache size using -Xscmx parameter

If you try to use -Xscmx to change the size of a cache after it has been created, the new size is ignored.

Dynamic cache update

Suppose that the HelloWorld class is modified to include another method (which simply prints ‘doing stuff’) and recompiled.

The existing .class file in the filesystem is replaced with the updated class, invalidating the version stored in the cache.

If you run the application again, you will see that the cache detects a newer version of the class is available, and marks the cached copy as *stale*.

The new class is read from disk and a copy stored in the cache, as shown in Figure 6-8.

```
                              QSH Command Entry

> java -Xshareclasses:name=myCache HelloWorld
  Hello world!
  doing stuff
  $
> java -Xshareclasses:name=myCache,printAllStats
...
3: 0xE0FF99BC CLASSPATH
        /home/rumsbya
3: 0xE0FF9984 ROMCLASS: HelloWorld at 0xE010DC10.!STALE!
        Index 0 in classpath 0xE0FF99BC
...
9: 0xE0FEFA84 ROMCLASS: HelloWorld at 0xE02306D0.
        Index 0 in classpath 0xE0FF99BC
...
# Stale classes    = 1
% Stale classes    = 0%

Cache is 14% full

Unable to create Java Virtual Machine.
$
```

Figure 6-8 Classes are marked as stale when a newer version is detected

6.2.3 Recommendations for shared classes

This section presents some recommendations you must consider before deploying shared classes. It is not wise to enable shared classes without first considering the applications that run on the system and how they are going to have to interact with the class cache.

Cache naming

If multiple users are running the same application, you may want them to use different caches. In this case, each user can use the default cache name, which incorporates the current user profile name to ensure uniqueness.

Or if you want a more meaningful unique name, you can use the %u modifier when specifying the cache name, which substitutes the user profile name.

If the same operating system group of users use the same application, you might want them to share the same cache to maximize the number of classes that are shared.

On i5/OS, by default, caches are only accessible by the user profile that created them. To share a cache between a group of users, they must belong to the same primary group profile.

The cache must also be created with the groupAccess suboption, as shown in Figure 6-9.

```
                              QSH Command Entry

> java -Xshareclasses:name=groupTest,groupAccess HelloWorld
  Hello world!
  $

===> ____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
 F3=Exit   F6=Print F9=Retrieve F12=Disconnect
 F13=Clear F17=Top  F18=Bottom  F21=CL command entry
```

Figure 6-9 The groupAccess suboption allows different users in the same group to share a cache

If multiple operating system groups are running the same application, the %g modifier can be added to the cache name. Each group running the application then gets a separate cache.

Number of caches to use

It is often the case that different applications (those in which not many classes are common) benefit more from sharing a single larger cache than using multiple smaller caches. With a single larger cache, you are still sharing the bootstrap classes between all JVMs, which is a worthwhile saving in the majority of cases.

It is recommended, therefore, to keep the number of caches to a minimum to maximize the opportunities for sharing. The exception is when separate caches are required for maintenance purposes or to provide a degree of isolation. This may be desirable, for example, when testing a new version of an application.

Even in these situations, it might be more efficient to create a unique modification context for the application in the single cache, rather than to create a separate cache. Refer to “Modification contexts” on page 68 for information about creating a modification context.

Use JAR or zip files, not .class files

The classes in the cache are automatically kept up-to-date by constant checking to ensure the time stamp information for the class matches the filesystem. If the version of the class in the filesystem is found to be newer, the copy in the cache is marked as *stale* to prevent other JVMs loading the old class from the cache. This checking is an unavoidable source of overhead which varies depending on whether the classloader loads most classes from JAR or zip files, or if the majority of classes are read from .class files.

JAR and zip files are handled differently by the cache compared to .class files. When a classloader loads a class from a JAR or zip file, it can lock the file, preventing it from being updated. Therefore, checking is not performed as often for .class files, which because they are not locked by the classloader, might disappear from the filesystem at any time (or a newer version of the .class file could open, invalidating the cached version). The implication is that cache operation is more efficient when classes are loaded from a JAR or zip file, compared to a .class file. Therefore, when designing Java applications to exploit shared classes, consider packaging classes into JAR or zip files rather than a number of stand-alone .class files.

Cache size

The shared classes cache is a fixed size, although this size is configurable when the cache is created. A full cache does not affect the functionality of any JVMs that are connected at the time, or that subsequently connect to it. However if the cache becomes full it might not give optimal efficiency or performance. Classes that are already stored in the cache continue to be loaded from the cache. However, no new classes can be stored in the cache and therefore they are read from disk into the JVM's private process memory.

It is also worth knowing that if a class in a full cache is marked as stale, there is going to be no space to store updated metadata about the class. Classes are pessimistically marked stale and a new piece of metadata is stored if it eventually proves not to be stale, *redeeming* the class. If there is no space in the cache to store the updated metadata, the cache cannot redeem any stale classes, and they are read from disk even though the class has not changed from the version originally stored in the cache.

We suggest, therefore, to create the cache with sufficient size for the applications that use it so that it does not fill completely. You can do this by running the application initially using a large cache size (set using the -Xscmx command-line parameter) then using the printStats utility suboption to determine how much class data was stored. You must perform this after the application has terminated because classes are loaded throughout the JVM lifetime, even during the shutdown process. You must add a small amount to the value shown in printStats output for contingency, as shown in Figure 6-4 on page 56. A good starting number is 10% of the cache size, but you still have to test your application to make sure you have sufficient cache size.

6.3 Shared classes and application performance

In addition to understanding the mechanism of the shared classes, you must consider the type of application you are running with shared classes.

6.3.1 When to use shared classes

Shared classes leads to the greatest performance benefit when multiple JVMs run concurrently on a system, or if JVMs restart frequently. For example, WebSphere Application Server V6.1 uses shared classes by default. If you run multiple instances of WebSphere Application Server on one server in a vertical cluster, you might notice the faster startup times and less virtual memory required by each JVM.

If you are in a test or development environment and are restarting the JVM often to test different tuning strategies, shared classes lead to faster JVM restarts.

6.3.2 Overheads associated with shared classes

The first JVM to run has to populate the cache so that classes can be shared. This adds a slight overhead to the startup time of the first JVM, varying by the number of classes present in the application. Expect up to 5% performance impact compared to a non-sharing JVM. However, this initial cost can be significantly smaller compared to the time saved by subsequent JVMs that load classes from the cache instead of from disk.

Class cache takes some space available in a JVM process. This is one of the reasons why it is said that the practical memory limit for a Java heap is 2.5 GB to 3 GB. For WebSphere Application Server, the maximum memory available for a Java heap is approximately 2.5 GB.

Shared classes and garbage collection

Shared classes do not affect the operation of the garbage collector, or the eligibility of classloaders to be garbage collected. Classloaders which use shared classes may still be garbage collected if there are no references to the classloader instance from anywhere else in the JVM. Class data stored as a ROMClass in the cache is never garbage collected, even if the classloader that loaded it into the cache is collected.

6.3.3 Performance gains with shared classes

We estimate the JVM startup time for a system using shared classes improves 10% to 40% depending on the application. The more JVMs running concurrently, the greater the performance benefits. If the cache is being shared by similar applications, there is more scope for sharing. Improved startup time and reduced memory footprint make a compelling case for enabling shared classes in systems using multiple JVMs or where JVMs are frequently restarted.

6.4 Shared Classes Helper API

IBM has provided a Shared Classes Helper API to allow developers to integrate class-sharing support into their own classloaders. Although the details of managing shared classes are largely handled by the JVM for you, if you are using the Helper API you have to manage finding and storing classes in the cache using the Helper API.

Note: This section applies to you only if you plan to write your own classloader.

6.4.1 Helper API overview

The Helper API classes are in the com.ibm.oti.shared package, contained in the vm.jar file under the following directory:

```
/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib
```

Tip: It is recommended extending java.net.URLClassLoader to take advantage of shared classes support built-in to this class. Otherwise you have to use the helper API to add shared classes support to your custom classloader.

For your classloader to share classes, it must obtain a SharedClassHelper object from a SharedClassHelperFactory. The SharedClassHelperFactory is a singleton object returned by calling the following static method:

```
com.ibm.oti.shared.Shared.getSharedClassHelperFactory()
```

This method returns a factory if shared classes is enabled in the JVM. Otherwise it returns null.

The SharedClassHelper gives the classloader a simple API for finding and storing classes in the class cache to which the JVM is connected. After it is created, the SharedClassHelper belongs to the classloader that requested it and can only store classes defined by that classloader. The classloader and SharedClassHelper have a one-to-one relationship.

Calling the static findHelperForClassLoader() method of the factory returns the SharedClassHelper object for the specified classloader, if one already exists. If the classloader is garbage collected, its SharedClassHelper object is also garbage collected.

Types of SharedClassHelper

There are several types of SharedClassHelper that you can use depending on the classpath format the classloader has to work with. These are defined as subinterfaces of SharedClassHelper:

- SharedClassTokenHelper

  Used by classloaders that require complete control over cache contents. The classloader stores and finds classes using an arbitrary String token generated by the classloader. This provides greatest flexibility. However, the classloader requires additional logic to generate and keep track of the tokens so that it can retrieve classes from the cache at a later time.

- SharedClassURLHelper

  Used by classloaders that do not have the concept of a classpath. The classloader stores and finds classes using a URL path such as a file system location. For example as follows:

  ```
  file:///home/rumsbya/javaclasses
  ```

- SharedClassURLClasspathHelper

  Used by classloaders that have the concept of a classpath. The classloader stores and finds classes using a URL classpath. Each classpath is composed of one or more URLs.

> **Restriction:** You can find classes stored by a SharedClassURLHelper using a SharedClassURLClasspathHelper, and the opposite also applies. However, you can find classes stored using a SharedClassTokenHelper only by using a SharedClassTokenHelper.

Classes stored in the cache by a SharedClassTokenHelper are *not* dynamically updated if the class changes in the filesystem. The reason is that the Tokens used to store classes have no meaning to the cache, therefore, it has no way of obtaining version information to use for comparison. Therefore, if you require cached classes dynamically updated to reflect any changes, use a SharedClassURLHelper or SharedClassURLClasspathHelper to store classes instead.

6.4.2 Helper API usage

Regardless of the SharedClassHelper API your classloader is using, two key functions are provided to the classloader for working with the cache:

- findSharedClass()

  The delegation model must continue to be followed by user-defined classloaders. The classloader must continue to ask its parent for a class before checking the cache, by calling super.loadClass().

  If this returns null, indicating no parent classloader has loaded the class already, then the classloader must call findSharedClass() on the SharedClassHelper. This returns the ROMClass from the cache as a byte array, or null if the class is not in the cache.

  Due to the distinction between ROMClasses and .class files, the byte array returned by findSharedClass() is not the actual class bytes. The classloader cannot instrument or modify these raw bytes in the same way that is possible with class bytes loaded from disk, because after they are stored in the cache, they are read-only. The byte array returned by this method must be passed to defineClass() in order to define the class for the current JVM and return it as a java.lang.Class object.

  If findSharedClass() returns null, the class is not in the cache, or the cache contains a *stale* copy, which means that the cache is aware a newer version of the class is available.

In either case, the classloader must retrieve the class bytes from disk, define the class using defineClass(), then pass the resulting java.lang.Class object to storeSharedClass().

- storeSharedClass()

  This Helper API method can be used by the classloader to store java.lang.Class objects in the cache so that they can be shared between JVMs. The classloader must define the class first by calling defineClass().

Important: Since the bytes returned from findSharedClass() are not actual class bytes, the classloader must never try to define a class from these bytes and store the resulting class in the cache using storeSharedClass(). Only classes loaded from disk must be passed to storeSharedClass().

6.4.3 Shared Classes Helper API example

Example 6-1 shows a partial listing for a simple user-defined classloader that makes use of the Shared Classes Helper API to work with the cache.

Example 6-1 Partial source listing for YourcoClassLoader.java

```
import sun.misc.URLClassPath;
import java.net.URL;
import java.lang.ClassLoader;
import java.io.IOException;
import com.ibm.oti.shared.*; // shared classes support import statement

public class YourcoClassLoader extends ClassLoader {
   private URLClassPath ucp; // The search path for classes
   private SharedClassURLClasspathHelper helper; // Shared classes helper object
   int storeAt = 0; // current index into the cache used for storing new classes

   // Wrapper class used for finding shared classes
   private class SharedClassIndexHolder implements SharedClassURLClasspathHelper.IndexHolder {
      int index;

      public void setIndex(int index) {
         this.index = index;
      }
   }

   protected void addURL(URL url) { // manage the classpath
      ucp.addURL(url);
   }

   private void initSharedClassesSupport(URL[] initialClassPath) {
      SharedClassHelperFactory factory = Shared.getSharedClassHelperFactory();

      if (factory != null) {
         try {
            this.helper = factory.getURLClasspathHelper(this, initialClassPath);
         } catch (HelperAlreadyDefinedException ex) {
            ex.printStackTrace();
            }
      }
   }
```

```
   public YourcoClassLoader(URL[] urls, ClassLoader parent) {
      super(parent);
      if(Shared.isSharingEnabled()){
         initSharedClassesSupport(urls);
      }
      ucp = new URLClassPath(urls, null, helper);
   }

   protected Class<?> findClass(final String name){
      // Try to find the class from the shared cache using the class name
      Class clazz = null;

      if (Shared.isSharingEnabled()) {
         SharedClassIndexHolder indexHolder = new SharedClassIndexHolder();
         byte[] sharedClazz = helper.findSharedClass(name, indexHolder);

         if (sharedClazz != null) {
            clazz = super.defineClass(name,sharedClazz,0,sharedClazz.length);
            return clazz;
         }
         else{
            byte[] newClazzBytes = ... // read class bytes from disk
            Class newClazz = super.defineClass(name, newClazzBytes,0,newClazzBytes.length);
            helper.storeSharedClass(newClazz, storeAt);
            storeAt++;
         }
      }
      else{ // shared classes disabled, load class from disk, no cache
         ...
      }
      return clazz;
   }

   public final synchronized Class loadClass(String name, boolean resolve) throws
ClassNotFoundException{
      return super.loadClass(name, resolve);
   }
}
```

6.4.4 Shared Classes Helper API Javadoc

The Shared Classes Helper API documentation is shipped as a zip file with IBM Technology for JVM on most platforms. i5/OS users can obtain this zip file by visiting the FTP site for this IBM Redbook. Refer to Appendix E, “Additional material” on page 199 for more information about accessing the FTP site.

The Shared Classes Helper API documentation is available in the SharedClasses folder of the extracted apidoc.zip file. Refer to Appendix E, “Additional material” on page 199 to download this file.

6.5 Security considerations

Using shared classes is no less secure than each JVM having its own copy of a class. Shared classes are subject to the same bytecode verification as non-shared classes. Care has to be taken if you are using bytecode instrumentation of classes. Refer to 6.6, “Bytecode modification” on page 68.

Using the shared classes feature is subject to two levels of security:

- Operating system security
- Java security (only if you are using a Java SecurityManager)

6.5.1 Operating system security

Operating system security can affect shared classes in several ways. When enabling shared classes, the first JVM to start asks the operating system to create several objects:

- A shared memory area for the cache
- A shared semaphore to coordinate access to the cache
- The following directory and two files underneath this directory for each cache: `/tmp/javasharedresources`

All of these operations are subject to standard operating system security. Therefore any could conceivably fail for a variety of reasons, such as insufficient authority or even insufficient shared memory/disk space.

The cache is created with user access by default unless the groupAccess suboption of -Xshareclasses is used, as shown in Figure 6-9 on page 61. This means that by default, only JVMs started under the same user profile as the JVM which initially created the cache, is able to use the cache.

If JVMs are started under different user profiles, these profiles have to share the same primary group profile in order to share the same cache. Usually however JVMs start under the same user profile, therefore, it is rare that you have to use the groupAccess suboption to share the same cache. For example, WebSphere Application Server, by default, runs all application servers under the QEJBSVR user profile.

6.5.2 Java security

If a Java SecurityManager is in use and the application uses custom classloaders, then these must be granted shared class permissions before they can share classes. You modify the java.policy file to add the shared class permissions using the class name of the custom classloader, or through specifying a wildcard. The type of access granted to the shared classes cache is one of “read”, “write”, or “read/write”.

Example 6-2 shows granting “read/write” shared classes permissions to all classloaders in the com.yourco.customclassloaders package.

Example 6-2 Extract from java.policy file showing shared class permission

```
permission com.ibm.oti.shared.SharedClassPermission
        "com.yourco.customclassloaders.*", "read,write";
```

Restriction: You cannot use a SecurityManager to restrict or modify the shared classes permissions of the Bootstrap, Extension, or System classloaders supplied with IBM Technology for JVM.

6.6 Bytecode modification

Runtime modification of Java class bytecode is being used increasingly often. You can use such *instrumentation* to introduce additional functionality into Java classes dynamically, in response to application state. For example, you can add sophisticated trace functionality to a class as required (and subsequently remove it when finished with it) while the application is still running. This reduces the overhead of having permanent trace code in all classes that has to be controlled by a global trace variable.

You can perform bytecode instrumentation by a variety of methods. A JVM Tools Interface (JVMTI) agent might be used to hook into the application, or the classloader itself may modify the class bytes before defining the class. Storing such classes in the cache without any indication they have been modified could cause major problems for JVMs that expect to retrieve the original class from the cache.

Fortunately, IBM Technology for JVM manages the cache so that it may be shared safely by both JVMs that use the modified class, and also those that do not.

Modification contexts

When starting the JVM, if you want to intentionally share or use a shared instrumented class, you can specify the appropriate *modification context*. A modification context is a user-defined descriptor which indicates the type of modification applied to the class bytecodes. Each modification context is associated with a private storage area where modified classes are stored. Specifying a modification context means that JVMs not using the modified class can safely use the same cache as those that are.

A cache can contain multiple modification contexts. You might have a modification context used for debugging, which is specified when starting an application that is causing problems. If you decide to enable debugging dynamically at runtime, any classes instrumented with debug code are safely stored in a separate area of the cache so that they cannot interfere with the unmodified version used by other JVMs. The application being debugged still gets the benefit of sharing any unmodified classes while it is running in debug mode.

Bytecode instrumentation might have been performed by a JVM Tool Interface agent, or by a user-defined classloader without using a JVMTI agent. If instrumentation is performed by a JVMTI agent, a modification context does not strictly have to be specified when sharing classes. The JVM can detect the modification and handle partitioning transparently.

Tip: It is recommended always specifying a modification context when sharing classes that have been modified by a JVMTI agent. Although not strictly required for safe sharing of modified and unmodified classes in the same cache, the JVM has to perform extra checks at runtime on modified classes not associated with a modification context. These checks impact performance slightly.

Important: If class modification was *not* performed by a JVMTI agent (for example, by a user-defined classloader) you *must* specify a modification context. If you do not, the cache is not partitioned and all JVMs connected to the same cache are going to use the modified version of the class, causing unexpected and undesirable results.

Modification context example

Modification contexts are specified by the user at runtime using the modified=context suboption of -Xshareclasses. If a modification context of the same name already exists in the specified cache, the JVM shares this context with other JVMs started under the same modification context. This allows you to intentionally share modified classes between multiple JVMs, for example, instrumented debug versions of the classes could persist across multiple debug runs of an application for efficiency.

We used the printAllStats suboption to view the modification contexts associated with a cache. Refer to “Interpreting printAllStats output” on page 57 for more information.

Figure 6-10 shows the HelloWorld application creating a cache called `myCache` and a modification context called `myModification1`.

```
                              QSH Command Entry

> java -Xshareclasses:name=myCache,modified=myModification1 HelloWorld
  Hello world!
  $
> java -Xshareclasses:name=myCache,printAllStats

  Current statistics for cache "myCache":

  1: 0xE0FFF8A4 CLASSPATH
  (modContext=myModification1)
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/vm.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/core.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/charsets.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/graphics.jar
...
1: 0xE0FFCD08 CLASSPATH
(modContext=myModification1)
        /home/rumsbya
1: 0xE0FFCCD0 ROMCLASS: HelloWorld at 0xE010DC10.
        Index 0 in classpath 0xE0FFCD08
  ...
```

Figure 6-10 Creating a new cache and modification context

The output in Figure 6-10 indicates that JVM 1 stored the class HelloWorld in the cache. The metadata about the class is stored at address `0xE0FFCCD0` and the class itself is written to address `0xE010DC10` in the cache. It also indicates the classpath against which the class is stored and from which index in that classpath the class was loaded, in this case from the first (and only) entry in the classpath at `0xE0FFCD08`.

If you run the application several more times, without modifying the class bytes, you can see that only one ROMClass is stored in the cache (at 0xE010DC10) and shared between all JVMs, as shown in Figure 6-11.

```
                              QSH Command Entry

> java -Xshareclasses:name=myCache HelloWorld
  Hello world!
  $
> java -Xshareclasses:name=myCache,modified=myModification2 HelloWorld
  Hello world!
  $
> java -Xshareclasses:name=myCache,printAllStats

  Current statistics for cache "myCache":

  1: 0xE0FFF8A4 CLASSPATH
  (modContext=myModification1)
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/vm.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/core.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/charsets.jar
          /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/graphics.jar
...
1: 0xE0FFCD08 CLASSPATH
(modContext=myModification1)
        /home/rumsbya
1: 0xE0FFCCD0 ROMCLASS: HelloWorld at 0xE010DC10.
        Index 0 in classpath 0xE0FFCD08
  ...
3: 0xE0FFC534 CLASSPATH
        /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/vm.jar
        /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/core.jar
        /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/charsets.jar
        /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/graphics.jar
...
3: 0xE0FF99BC CLASSPATH
        /home/rumsbya
3: 0xE0FF9984 ROMCLASS: HelloWorld at 0xE010DC10.
        Index 0 in classpath 0xE0FF99BC
...
5: 0xE0FF91D8 CLASSPATH
(modContext=myModification2)
        /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/vm.jar
        /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/core.jar
        /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/charsets.jar
        /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/graphics.jar
...
5: 0xE0FF6650 CLASSPATH
(modContext=myModification2)
        /home/rumsbya
5: 0xE0FF6618 ROMCLASS: HelloWorld at 0xE010DC10.
        Index 0 in classpath 0xE0FF6650
...
```

Figure 6-11 Content of the class cache

Note we also created a second modification context called `myModification2`. Figure 6-11 shows that despite the number of modification contexts that may exist, if the class bytes are identical there is only one ROMClass stored in the cache for each application that uses the HelloWorld class, in this case at address `0xE010DC10`.

Using modified bytecode with shared classes

Next we modify the bytes of the HelloWorld class on-the-fly to show that a different version of the class is stored in the cache, in the separate area defined by the modification context. As mentioned earlier, there are two broad ways of doing this, using a JVMTI agent or a user-defined classloader to instrument the class before it is defined.

For this example, we use the HPROF JVMTI agent. HPROF is a simple demonstration JVMTI agent supplied with IBM Technology for JVM. It can gather various data about an application, such as time spent in each method and heap activity. HPROF uses bytecode instrumentation depending on which options you specify at runtime. Not all options require the application to be instrumented.

When you run Java with HPROF, an output file is created at the end of program execution. This file is placed in the current working directory and is called java.hprof.txt. You can specify additional command line argument *file=<filename>* to specify a different file name.

For our purposes we want to modify the bytecodes, therefore we use the cpu=times option with HPROF. This inserts bytecodes into all method entry and exit points, therefore, clearly the bytecodes of the HelloWorld class are going to differ from the version in the cache after it is instrumented by HPROF. The command line and output are shown in Figure 6-12.

```
                                QSH Command Entry

> java -agentlib:hprof=cpu=times -Xshareclasses:name=myCache,modified=hprofcontext HelloWorld
  Hello world!
  Dumping CPU usage by sampling running threads ... done.
$
> java -Xshareclasses:name=myCache,printAllStats
...
1: 0xE0FFCD08 CLASSPATH
(modContext=myModification1)
        /home/rumsbya
1: 0xE0FFCCD0 ROMCLASS: HelloWorld at 0xE010DC10.
        Index 0 in classpath 0xE0FFCD08
...
7: 0xE0FF5E70 CLASSPATH
(modContext=hprofcontext)
        /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/vm.jar
        /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/core.jar
        /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/charsets.jar
        /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/graphics.jar
...
7: 0xE0FF32EC CLASSPATH
(modContext=hprofcontext)
        /home/rumsbya
7: 0xE0FF32B4 ROMCLASS: HelloWorld at 0xE022E4B0.
        Index 0 in classpath 0xE0FF32EC
...
```

Figure 6-12 Using HPROF

You can see that there are now two versions of the HelloWorld class in the cache, the original version at `0xE010DC10` and the instrumented version at `0xE022E4B0`.

7

Java Native Interface

This chapter discusses the Java Native Interface (JNI), the standard native programming interface for Java to invoke native methods. This chapter provides an overview of the JNI and discusses specific considerations for using the JNI in IBM Technology for JVM. This chapter is not intended to be a comprehensive tutorial on how to use the JNI, however, 7.5, “Further information about the JNI” on page 81 contains links to learning resources for using the JNI.

This chapter discusses the following topics:

7.1 Java Native Interface overview

The Java Native Interface (JNI) is the standard native programming interface for Java, which is integrated into the Java Development Kit (JDK). JNI allows Java applications to interact with applications and libraries that are written in other languages, such as C, C++, and RPG. In addition, the Invocation application programming interface (API) defined as part of the JNI allows you to embed a Java virtual machine (JVM) into your native application. This is useful in situations where you want to use functionality of a Java application or class from your native application.

JNI provides a standard way to interface with programs written in other languages. This allows you to maintain a single version of your native method libraries on that platform.

Note: If an application uses the JNI, by virtue of relying on native code it ceases to be a *100% pure Java* application. Therefore, if portability is important to you, investigate alternatives to JNI such as the IBM Toolbox for Java classes.

The JNI has two main purposes:

- It specifies a way to write Java native methods. Java native methods invoke code written in other languages native to the platform on which Java is running.
- It includes an Invocation API for embedding a JVM in native applications.

You can use JNI to write native methods to handle those situations when an application cannot be written entirely in the Java programming language. For example, you might have to use native methods and JNI in the following situations:

- The standard Java class library does not support the platform-dependent features that your application requires.
- You have a library or application that is written in another programming language and you want to make it accessible to Java applications.
- You want to implement a small portion of time-critical code in a lower-level programming language, such as C, and have your Java application call these functions.

Programming with the JNI framework lets you use native methods to perform many operations. You may use native methods to represent legacy applications or explicitly to solve a problem that is best handled outside the Java programming environment. The JNI framework lets your native method use Java objects in the same way that Java code uses these objects.

The JNI serves as the glue between Java and native applications.

Figure 7-1 shows how the JNI ties the native, or C, side of an application to the Java side.

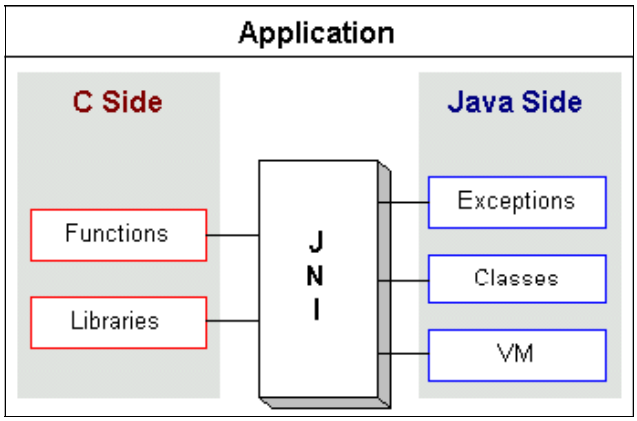


Figure 7-1 JNI links Java code to native code

A native method can create Java objects, including arrays and strings, and then inspect and use these objects to perform its tasks. A native method can also inspect and use objects that are created by Java application code. A native method can even update Java objects that it created or that were passed to it. These updated objects are available to the Java application. Therefore, both the native language side and the Java side of an application can create, update, and access Java objects and then share these objects between them.

Native methods can also call Java methods. Often you will already have developed a library of Java methods. Your native method does not have to "re-invent the wheel" to perform functionality that is already incorporated in existing Java methods. The native method, using the JNI framework, can call the existing Java method, pass it the required parameters, and get the results back when the method completes.

7.2 When must you use JNI?

C, C++, or RPG programmers who want to integrate with Java methods might want to start programming in JNI. JNI protects your code from unknowns, such as the vendor-specific VM extension that is used to integrate traditional language with Java. By conforming to the JNI standard, a native library (C, C++, or RPG) is given the best option to run problem free in any standard implementation of a JVM. The JNI specification does not impose any overhead or restrictions on any JVM implementation, including object representation, garbage collection scheme, and so on.

For the System i platform, several other alternatives to using the JNI are available including:

- IBM Toolbox for Java access classes such as Distributed Program Call and Data Queue support
- The Extensible Program Call Markup Language (XPCML) support available with the IBM Toolbox for Java

The IBM Toolbox for Java access classes and XPCML are easier to program than JNI. The advantage of using JNI is that both the calling program and the called program run in the same process (job) on the System i server, while some other methods, such as ProgramCall, might use a new process (job). This makes JNI calls faster at startup time and less resource intensive. However, using IBM Toolbox for Java access classes ensures the application remains *100% pure Java* for maximum portability.

For more information about the IBM Toolbox for Java and XPCML, refer to the System i Information Center at:

http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=/rzahh/rzahhxmlmain.htm

7.3 JNI considerations in IBM Technology for JVM

Be aware of these considerations and caveats before using JNI in IBM Technology for JVM. Also, there are some differences in requirements for native code used with JNI compared to Classic JVM.

7.3.1 Teraspace storage model

If you want to use IBM Technology for Java and have programs that use native methods, you must compile these programs with teraspace storage enabled. Because teraspace storage is not enabled by default, it is likely that you have to recompile. This is necessary because the Java object is in i5/OS PASE storage, which is mapped on top of teraspace storage, and a teraspace storage pointer is returned. Also, the JNI functions, such as GetxxxArrayRegion, have a parameter to a buffer where the data is placed. This pointer must point to teraspace storage to enable the JNI function in i5/OS PASE to copy the data into this storage. If you have not compiled your program with teraspace storage enabled and attempt to run the native method with IBM Technology for Java, you are going to receive the escape message:

```
MCH4443 (Invalid storage model for target program LOADLIB)
```

When you recompile a native method, use the following options:

```
TERASPACE(*YES) STGMDL(*TERASPACE) DTAMDL(*LLP64)
```

For more information see InfoCenter at:

http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=/rzaha/rzahateraspacemain.htm

7.3.2 Thread safety of i5/OS system functions

Not all i5/OS system functions are thread safe. Therefore, using the JNI support to invoke a C function that calls a non-thread safe i5/OS system API might cause unpredictable results to occur. The System API Reference lists the thread safe status of every system API. It is recommended consulting this reference before using any system API through the JNI support provided in IBM Technology for JVM. The System API Reference for the current V5R4 is available at:

http://publib.boulder.ibm.com/infocenter/iseries/v5r4/topic/apifinder/finder.htm

7.3.3 64-bit PASE native methods

In V5R4 IBM Technology for Java Virtual Machine runs in a 32-bit PASE. If you have 64-bit PASE native methods, you cannot use 32-bit IBM Technology for Java Virtual Machine to call these methods and must use Classic JVM instead.

When 64-bit IBM Technology for JVM is available in i5/OS, you are going to be able to call 64-bit PASE native methods.

7.3.4 Adopted authority

Basically, a method can adopt the authority of the owner of the program being run rather than the authority of the person running the program in order to accomplish some operation. System i platform has provided the adopted authority feature for Java programs since the JVM was part of the Trusted Computing Base in System Licensed Internal Code (SLIC).

As IBM moves to a Java implementation where the JVM is a user level program, Java can no longer provide this feature. In order to achieve the same functionality, the most common approach is to add an Integrated Language Environment (ILE) native method to the Java program that adopts equivalent user profile authority and performs the required operation.

This section discusses two scenarios where it refers to some attributes of the native objects. During runtime, an i5/OS program object identifies the adoption attributes of the invocation. One of these attributes is Allow/Drop Adopted Authority. In the scenarios we specify the value of this attribute in the parentheses. We also highlight a method which adopts authority.

Scenario 1

A Java method adopts authority immediately before calling a native method. See Figure 7-2.

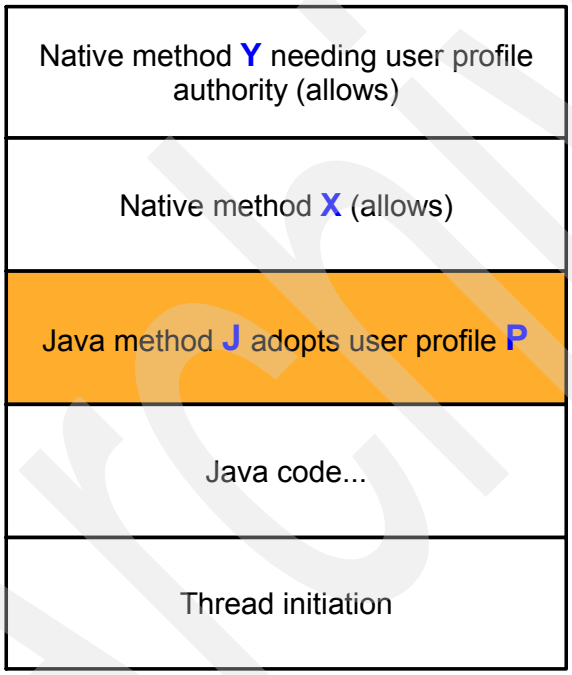


Figure 7-2 Scenario 1

In this example, Java method J is contained in a Java program which adopts user profile P and directly calls native method X. Native method X calls ILE method Y which requires the adopted authority. To run this Java program using IBM Technology for JVM, select one of the following two alternatives.

Alternative 1

One way to preserve authority adoption is to segregate native method X into a new service program. This new service program might then adopt user profile P which was previously adopted by Java method J. This alternative is shown in Figure 7-3.

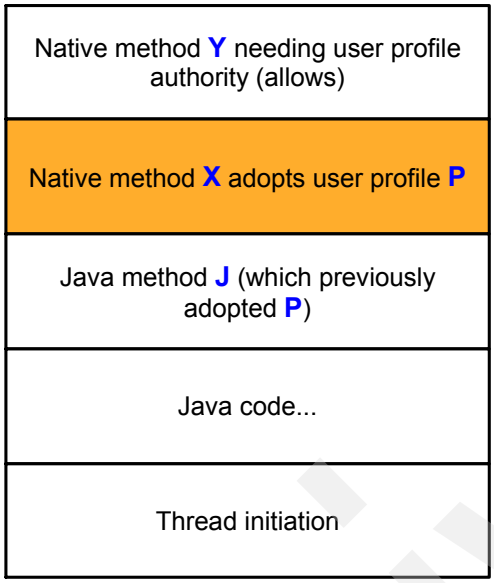


Figure 7-3 Scenario 1: alternative 1

Alternative 2

Another way to preserve user profile adoption is to create an entirely new native method N contained in a service program which adopts user profile P. This new method is called by Java method J and calls native method X. Java method J requires to be changed to call N instead of X, but native method X does not have to be changed or repackaged. Figure 7-4 shows alternative 2.

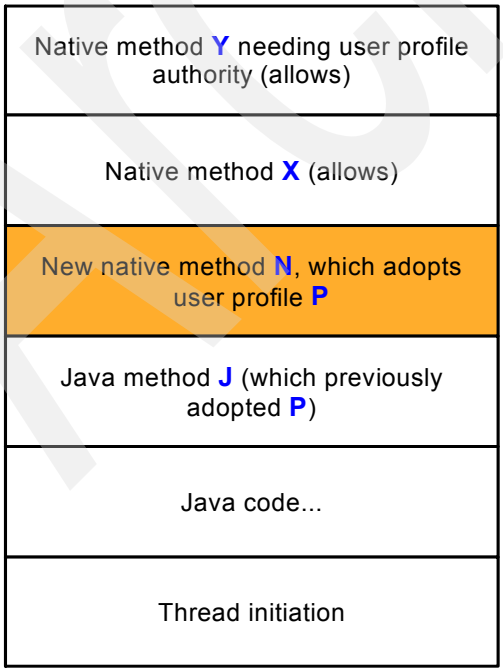


Figure 7-4 Scenario 1: alternative 2

Scenario 2

In the second scenario a Java method adopts authority and calls other Java methods before calling a native method. See Figure 7-5.

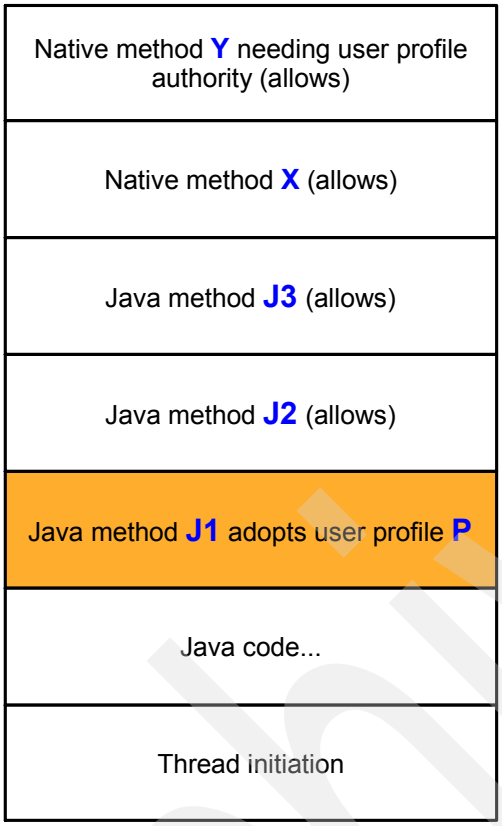


Figure 7-5 Scenario 2

In this example, a Java method *J1* is contained in a Java program which adopts user profile *P*. *J1* calls Java method *J2*, which calls *J3*, which calls native method *X*. Native method *X* calls ILE method *Y* which requires the adopted authority.

In order to preserve adopted authority in this case, a new native method *N* can be created. This native method is contained in a service program which adopts user profile *P*. Native method *N* would then use JNI to call Java method *J2*, which is unchanged. Java method *J1* would require to be changed to call native method *N* instead of Java method *J2*.

Figure 7-6 shows the alternative solution.

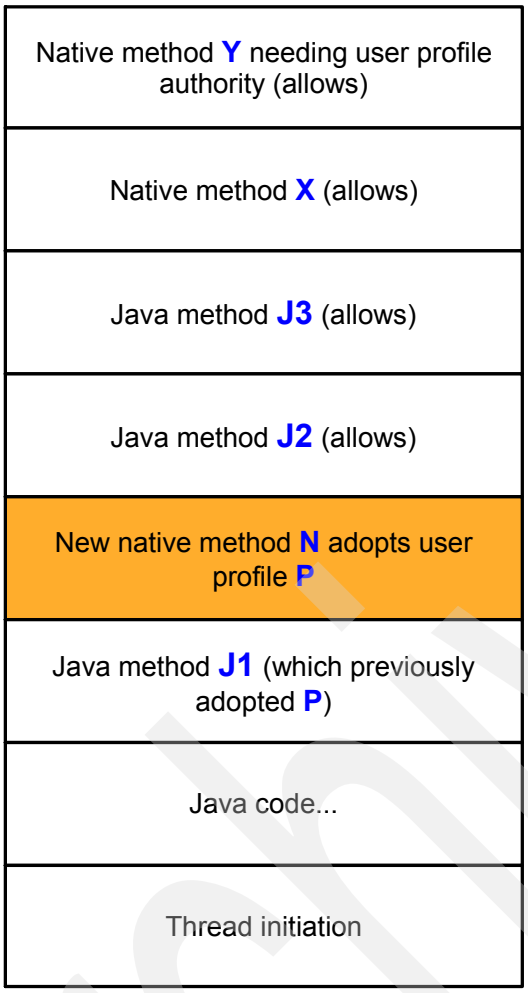


Figure 7-6 Alternative solution for scenario 2

7.3.5 Performance of native code

When making calls to ILE native code from Java applications through the JNI, you have to consider that most calls have to cross the boundary between the PASE (in which IBM Technology for JVM runs) and the SLIC. This adds significant overhead to the call. Consequently, making frequent calls to small ILE native procedures or system programs is the most unfavorable approach. The overhead of crossing the PASE-SLIC boundary is comparable or greater than the execution time of the ILE code. Performance of JNI call to ILE native code has been identified as a focus area for improvement.

On the other hand, calls to i5/OS PASE native methods from Java applications running under IBM Technology for JVM, perform better than in the Classic JVM. This is because IBM Technology for JVM runs in the same address space as i5/OS PASE native methods.

Attention: It is recommended that if a Java application must use JNI, then use i5/OS PASE native methods wherever possible in preference to other native methods. Performance of i5/OS PASE native methods is several orders of magnitude better than other native methods.

7.4 Diagnostic messages and files

When ILE native methods encounter problems, you are going to see messages in the job log. When IBM Technology for Java Virtual Machine or PASE native methods encounter problems, they dump diagnostic files into the Integrated File System (IFS). There are several types of these “core files”, including core.*.dmp, javacore.*.txt, Snap*.trc, and heapdump.*.phd. The files range in size from tens of KB up to hundreds of MB. In most cases, more severe problems produce larger files. The larger files can quickly and quietly consume large amounts of IFS space. Despite the space these files consume, they are useful for debugging purposes. When possible, you must preserve these files until the underlying problem has been resolved.

Notice, however, that the presence of dump files does not indicate that you have a JNI problem. These files can be generated for a wide variety of reasons.

7.5 Further information about the JNI

For more information about the JNI and its usage, the following resources are available:

- JNI 5.0 Specification (from Sun Microsystems, Inc.):

  http://java.sun.com/j2se/1.5.0/docs/guide/jni/index.html

- The System i Information Center:

  http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp

- Chapter 8 of the IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide:

  http://www-128.ibm.com/developerworks/java/jdk/diagnosis/

8

Analyzing JVM behavior

This chapter helps you to identify and solve certain kinds problems that you might have while running your application in IBM Technology for Java virtual machine (JVM). We take a practical approach in this chapter. This chapter identifies five major problem areas and demonstrates a process for finding the root cause of each type of problem. It demonstrates tools provided with the IBM Support Assistant where possible and how you can use them in the troubleshooting process. This chapter also presents a section on debugging Java applications using WebSphere Development Studio Client for iSeries.

This chapter covers the following topics:

8.1 JVM analysis overview

This chapter focuses on a *bottom-up* approach to troubleshooting problems. This means starting at the lowest level above the operating system, IBM Technology for JVM, the components of the system up to and including the application code itself are analyzed in turn to see if they are the cause of the problem.

There are many good resources for troubleshooting problems in a *top-down* manner. These approaches assume there is some high-level symptom which is being observed or reported by end users. For example, a Web-facing application might be timing out in a Web browser.

Troubleshooting in a top-down manner initially focuses on the application itself. If there is no evidence of a problem with the application, the next stage is to troubleshoot the middleware layer, for example WebSphere Application Server and associated components such as IBM DB2®.

If you suspect the issue might be with WebSphere Application Server, it is particularly recommended consulting the IBM Redbook *WebSphere Application Server V6 Problem Determination for Distributed Platforms*, SG24-6798.

Using a bottom-up approach to troubleshooting is suited to cases where there is some visible symptom occurring at a low level, typically this would be some anomaly in the JVM itself. For example, the JVM job might disappear unexpectedly and no longer be visible through WRKACTJOB. Or you might be seeing a memory leak in the form of heap exhaustion. Therefore, the first stage is to make sure there is no issue with the JVM itself causing the heap exhaustion.

If you find no evidence that the problem is with IBM Technology for JVM, then the next stage would be to troubleshoot the middleware layer, for example WebSphere Application Server. Logically, the next step after this would be to troubleshoot the application itself if no cause was found for the problem in the middleware layer.

Using a bottom-up approach does not guarantee faster problem resolution. You might eventually discover for instance that the application is making inefficient use of the heap by preserving object references longer than is required or instantiating many unnecessary objects. In this case, although the problem is with the application, the most visible symptom was the heap running out of space at the JVM level.

In some situations you have to contact IBM Support in order to identify and fix the problem. However, by following the recommendations in this chapter you can significantly reduce the amount of time required for IBM to provide a fix or workaround.

The five major problem that this chapter covers are:

- JVM does not start.
- JVM crash.
- Application becomes unresponsive.
- Application performance issues.
- Memory leak.

Figure 8-1 shows a diagram for using the bottom-up approach in troubleshooting IBM Technology for JVM.

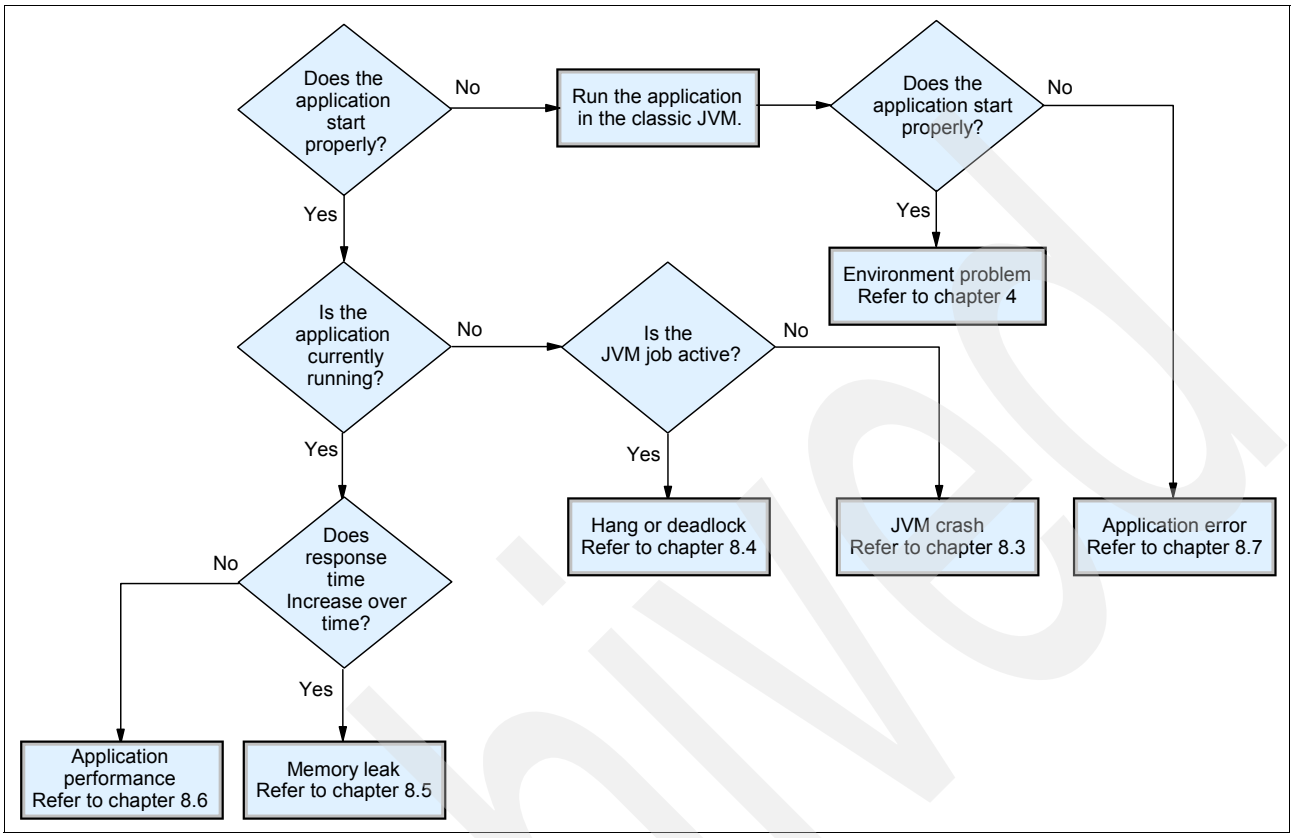


Figure 8-1 Basic problem determination steps for IBM Technology for JVM

8.2 JVM does not start

If you have an application using IBM Technology for JVM and it does not start, you have to isolate the problem into one of the following areas:

- Environment problem
- JVM problem
- Application problem

8.2.1 Quick fix

Before you start debugging a problem, it is recommended that you try the following tasks:

- Switch to Classic JVM
- Check group program temporary fix (PTF) level for Java

Use the Classic JVM

A good test is to run the application under the Classic JVM. Refer to 3.3, “Basic configuration” on page 23 for information about configuring i5/OS to use the Classic JVM. Make sure you use the correct Java version:

```
-Djava.version=1.5
```

If the JVM does not start with the Classic JVM, then the issue might be an environment problem. However, if the application runs properly under Classic JVM, then there might be an issue with IBM Technology for JVM. In either case, you are going to be closer to resolving the issue because IBM Support has an understanding of whether it is likely a JVM or environment issue and can apply more specific troubleshooting techniques.

Check PTF level

Check the fix level for Java by using the WRKPTFGRP CL command. Compare your level with what is currently available at the following Web site:

`http://www-912.ibm.com/s_dir/slineOO3.NSF/GroupPTFs?OpenView&view=GroupPTFs`

If there is a newer fix level you must order and install it.

8.2.2 Environment problem

If your program executes when using the Classic JVM you could have a problem with the setup of the IBM Technology for JVM environment. Try to check if the sample **Hello** program runs? Refer to 3.2.3, “Verifying the installation” on page 22 for instructions.

If a simple program like **Hello** runs properly, you must investigate what properties and environment variables are being set when you invoke your program.

Refer to Chapter 3, “New user guide” on page 19 for more information about setting properties and environment variables for IBM Technology for JVM.

8.2.3 JVM problem

If the sample **Hello** program does not run, there is likely a problem with the installation of IBM Technology for JVM or one of the prerequisite products such as i5/OS PASE. Refer to 3.2, “Installing IBM Technology for JVM” on page 20 for more information.

8.2.4 Application problem

If the Hello program runs in both JVMs, but your application fails to start using either Classic JVM or IBM Technology for JVM, the problem is likely specific to the application and must be debugged as such.

8.3 JVM crash

Abnormal termination of a JVM is known as a *crash*. When a crash occurs, your application is no longer running due to the JVM termination. A crash is characterized by the JVM job no longer being visible through WRKACTJOB.

Depending on the environment where you run an application (production or development), you might apply a different set of steps for resolving a crash. In production environment you might want to restart an application as soon as possible and then analyze the cause of a crash. In this case, you have to look at 8.3.2, “Quick fix” on page 88 as your first step.

More generally, the first step in analyzing a crash must be to check any log file and also look for javacore files. In many cases (especially when it is not a true crash of the process) these log files give an indication of what went wrong.

8.3.1 Causes of a crash

Crashes can occur for a variety of reasons, although most can be classified under three broad categories:

- Environment problem
- JVM problem
- Application problem

Environment problems might include running out of memory, incorrectly setting environment variables, and so on. Often problems in the environment involve some degree of user control, while JVM problems might not be resolvable by the user beyond implementing a temporary workaround.

JVM problems might include defects in the JVM itself or corrupt data in the JVM.

Application problems might include crashes due to poorly-written Java Native Interface (JNI) or error-handling code.

Crashes might involve a certain degree of predictability. They might occur whenever a certain set of conditions occur, or they might seemingly occur at random. For example, if a certain transaction keeps crashing the JVM at the same point, then there is a good chance either the data or methods associated with that transaction are causing the problem.

Crashes and native code

If your application loads any custom native libraries, for example a JVMTI agent to profile the application, try running the application without loading these libraries if possible. If the JVM does not crash, it suggests a problem with the native library.

You can check the Javadump file generated at the time of the crash (refer to “Locating a Javadump file” on page 88) to confirm if the problem is in your native code. The file has the following format:

```
javacore.DDDDDDDD.nnnnnn.nnnn.txt
```

Here `DDDDDD` is the date, and `nnnnnn.nnnn` is a time stamp.

It is recommended that you perform this before contacting IBM Support. The Javadump file shows which library the crash occurred in, as shown in Example 8-1.

Example 8-1 Extract from Javadump showing faulty library caused a General Protection Fault

```
------------------------------------------------------------------------
0SECTION       TITLE subcomponent dump routine
NULL           ===============================
1TISIGINFO     Dump Event "gpf" (00002000) received
1TIDATETIME    Date:                 2006/10/04 at 17:57:02
1TIFILENAME    Javacore filename:
/home/rumsbya/jars/javacore.20061004.175702.9689.txt
NULL
------------------------------------------------------------------------
0SECTION       GPINFO subcomponent dump routine
NULL           ================================
2XHOSLEVEL     OS Level         : OS400 5.4
2XHCPUS        Processors -
3XHCPUARCH       Architecture   : ppc
3XHNUMCPUS       How Many       : 2
NULL
```

```
1XHEXCPCODE    J9Generic_Signal_Number: 0000000000000004
1XHEXCPCODE    Signal_Number: 000000000000000B
1XHEXCPCODE    Error_Value: 0000000000000000
1XHEXCPCODE    Signal_Code: 0000000000000001
1XHEXCPCODE    Handler1: 0000008000587708
1XHEXCPCODE    Handler2: 000000800063AF00
NULL
1XHEXCPMODULE  Module: /home/rumsbya/crashlib.so
...
```

In Example 8-1 because the crash occurred in a custom library which is not part of the JVM, you must investigate any recent changes to the library which might have introduced a bug. If you have a previous version that was known to work, you could revert to using that while debugging the new library.

Locating a Javadump file

You can find this file using the following approach:

- The location specified by the *IBM_JAVACOREDIR* environment variable if set
- The current working directory of the JVM processes
- The location specified by the *TMPDIR* environment variable, if set
- The */tmp* directory
- If the Javadump cannot be stored in any of the previous locations listed, then it is put in to STDERR

8.3.2 Quick fix

If the JVM crashes, it is important to get the application up and running again to reduce the impact on the business and end users. It might be possible to workaround the crash using one of a variety of methods. Doing so is in your best interests because it reduces the impact of the problem to end users and your business. A temporary workaround is also going to “buy some time” to allow you to work with IBM Support on a permanent resolution.

If the crash is occurring in a test environment rather than production, then the importance of a quick fix might be lessened. However, the approaches that follow are still useful in narrowing the scope of where the issue is occurring and allows more specific troubleshooting steps to be followed. This particularly applies in cases where IBM Support are involved.

Use the Classic JVM

A good test is to run the application under the Classic JVM (refer to 3.3, “Basic configuration” on page 23 for information about configuring i5/OS to use the Classic JVM).

If the Classic JVM also crashes, then the issue might be an environment problem. However if the application runs properly under Classic JVM, then there might be an issue with IBM Technology for JVM. In either case, you are closer to resolving the issue because IBM Support has an understanding of whether it is likely a JVM or environment issue and can apply more specific troubleshooting techniques.

Disable the just-in-time compiler

The Java runtime environment can be viewed as two basic components, which are closely bound:

- The JVM itself
- The just-in-time (JIT) compiler

The JIT compiler generates native code from Java bytecodes. The native code is then executed directly, which improves performance. The JIT compiler is enabled by default. For further information about the JIT compiler, refer to “Just-in-time compiler” on page 6.

Although considered very rare, a failure in the JIT compiler might cause the JVM to crash. For this reason, if the JVM fails, it is useful to rule out the JIT compiler as a possible cause early in troubleshooting.

Therefore, it is recommended that you run the application with the JIT compiler disabled. To disable the JIT compiler, start the application with the -Xint option specified on the command line. For example, issuing the following command from a Qshell prompt causes the HelloWorld application to be run in purely interpreted mode:

```
java -Xint HelloWorld
```

If the application then runs properly, the issue mighty be related to the JIT compiler. In this case it is recommended that you contact IBM Support to report the failure. You can refer to the IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide for instructions on how to identify the specific method causing the JIT compiler failure. The IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide is available on the Web at:

http://www-128.ibm.com/developerworks/java/jdk/diagnosis/

Identifying the failing method means you can selectively disable compilation of just that method as opposed to all methods. The benefit of applying a more granular approach and disabling JIT compilation of just the failing method is that the vast majority of methods can still be compiled into native code. As a result, you only lose a negligible fraction of JVM performance in comparison to using interpreted mode only.

For example, issuing the following command from a Qshell prompt prevents JIT compilation of just the Math.max() method when the HelloWorld application runs:

```
java -Xjit:exclude={java/lang/Math.max(II)I} HelloWorld
```

This method, instead, is executed in interpreted mode, bypassing the JIT compiler. All other methods are eligible for JIT compilation.

8.3.3 If application runs properly under Classic JVM

There might be an issue with IBM Technology for JVM. You can contact IBM Support to report the issue and continue to use the Classic JVM as a temporary workaround. You might also wish to do some basic troubleshooting yourself, which might lead to a workaround under IBM Technology for JVM.

Heap differences

One case where the application might run properly under the Classic JVM, but not under IBM Technology for JVM, is where the JVM runs out of memory. Because of the approximate 3 GB heap size limit with the 32-bit IBM Technology for JVM, the application might not fit within the 32-bit address space. In this case you must continue to run the application under Classic JVM until such time that the 64-bit IBM Technology for JVM becomes available.

If you are experiencing crashes in IBM Technology for JVM but not Classic JVM, you must increase the maximum heap size as much as possible. If the application still uses all the heap and crashes, then it might be that the application’s heap requirements are too large for IBM Technology for JVM.

The other possibility is a memory leak. Due to the smaller maximum heap size, the effect of a leak is more pronounced in IBM Technology for JVM. With the Classic JVM an out of memory condition could take longer to occur or might not occur at all, because of the unlimited maximum heap size. Refer to 8.5, “Investigating a suspected memory leak” on page 111 for information about troubleshooting a memory leak.

8.3.4 If the Classic JVM also crashes

If the Classic JVM also crashes, then there might be an environment issue. The application might be exhausting the heap, which could occur for several reasons:

- Memory is leaking
- The maximum heap size has been set too small for the application’s requirements

A memory leak might be a more likely cause in this scenario where both JVMs crash, therefore, refer to in 8.5, “Investigating a suspected memory leak” on page 111 for guidance.

Try specifying a larger maximum heap size using the -Xmx command line parameter. For example, the following uses a maximum 3 GB heap size instead of the default 2 GB under IBM Technology for JVM:

```
java -Xmx3g HelloWorld
```

However, if the application crashes with an unlimited heap under Classic JVM, specifying a larger maximum heap size under IBM Technology for JVM is unlikely to alleviate the problem.

8.3.5 Identifying the cause of a crash

Reviewing the Javadump file (javacore*.txt) created during a crash indicates in which library the crash occurred. Refer to “Crashes and native code” on page 87 for an example.

It is straightforward from the library name and path whether the crash occurred in a JVM library or a custom library that might be used by an application. JVM libraries typically have a libj9 prefix and are located in the following directory by default:

```
/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib
```

As explained earlier, if the crash is in a custom library, this has to be debugged by the provider of the custom library. Crashes in JVM libraries must be reported to IBM Support for further investigation.

Regardless of whether the crash occurs in a JVM or custom library, using debugging tools such as those provided with WebSphere Development Studio Client for iSeries can help isolate the area of application code which triggers the crash. This assists with the problem isolation and determination effort. Refer to 8.7, “Application debugging” on page 155 for more information about using WebSphere Development Studio Client for iSeries to do application debugging.

8.4 Unresponsive application

This section outlines possible causes of an unresponsive application and how to troubleshoot a hang. We present the ThreadAnalyzer tool as an aid to identifying deadlocks and analyzing threads, both in WebSphere Application Server environments and stand-alone Java applications. We also describe how WebSphere Development Studio Client for iSeries can be used to investigate a suspected deadlock scenario.

8.4.1 Overview of a hang

An application that has stopped responding, but whose JVM is still running, is considered to have hung. A hang is differentiated from a crash in that a hung JVM is displayed in the WRKACTJOB output.

A hang can occur due to:

- Deadlock
- Infinite loop (spin)/CPU 'hogging'
- Timeout
- Blocked on resource
- Memory constraints

8.4.2 Hangs in applications running on WebSphere Application Server

If the application is running on WebSphere Application Server rather than as a 'stand-alone' Java application, then there are higher-level WebSphere Application Server-specific troubleshooting tools and techniques that you might wish to try first. The troubleshooting steps presented in this section are focused toward stand-alone Java applications and do not take account of advanced Reliability, Availability, and Serviceability (RAS) features built-in to WebSphere Application Server. For example, WebSphere Application Server includes a *hang detection* option which is enabled by default. This monitors managed threads and reports in the log when a thread has been executing too long within a unit of work, because such a thread may have hung.

If you plan to contact IBM Support about a hang issue with WebSphere Application Server, IBM Support Assistant can be used to collect the "MustGather" data required by IBM Support. Refer to "Submitting a problem report to IBM" on page 187 for more information.

8.4.3 Quick fix

Although restarting the JVM alleviates the problem, if the application runs long enough, the hang conditions are likely to occur again if no corrective action is taken. Therefore, if a hang occurs, it is recommended that you take some time to review this chapter and try some of the troubleshooting steps that are appropriate to your particular situation.

It is rare that a hang is due to an underlying JVM error. In addition to a JVM error, a hang can occur due to a poorly-written application or problem with the system environment. For example, if threads are not properly synchronized, deadlock can arise if resources are shared between multiple threads and contention occurs.

It is likely that a hung JVM has to be restarted because it is unlikely to make any progress in such a degraded state. If the hang occurs in a test environment, however, you might want to wait to determine if any progress is made if the application is left long enough.

Running the application under Classic JVM is a valid troubleshooting step. If the hang occurs in Classic JVM, then the application might be at fault. If no hang occurs, then there might be an issue with IBM Technology for JVM.

In situations where the hang occurs only under IBM Technology for JVM, it is recommended that you run the application with the JIT compiler disabled. Refer to "Disable the just-in-time compiler" on page 88 for information about how to disable the JIT compiler.

If the hang still occurs with the JIT compiler disabled, you have at least ruled out the JIT compiler as a possible cause. At this stage, because the Classic JVM and IBM Technology for JVM are exhibiting different behavior, you might want to engage IBM Support which is going to supply more specific troubleshooting steps.

For situations where the hang occurs under both JVMs, you must investigate the hang further to determine if it is an issue with the application or the environment. If the application tends to run for a relatively long period before hanging, a temporary workaround to “buy some time” for further investigation might be to schedule a periodic restart of the application. If you can identify with some degree of confidence how often or how long after startup the application hangs, you can schedule a restart before the expected window where the hang could occur, to reset the application environment and therefore reduce the likelihood of the hang occurring.

Under any circumstances you must take a javadump (refer to “Generating the Javadump” on page 93) before ending or restarting the JVM. Even if you do not intend to look at it, the javacore can be very helpful to IBM support.

8.4.4 Troubleshooting a hang

This section demonstrates methods and tools which you can use to troubleshoot a hang. Typically, you would refer to this section if you have a hang that occurs under both Classic JVM and IBM Technology for JVM, in which case the source of the hang is more likely to be related to the application or environment. We present an approach for each of the major causes of hangs.

In which order you try the troubleshooting steps depends on the symptoms you are seeing and your preference for using system tools or the ThreadAnalyzer tool with IBM Support Assistant. It is worth mentioning that the system tools do not readily help you detect hangs due to deadlock. In this case it is recommended that you capture a Javadump which you can use as input to ThreadAnalyzer for deadlock detection which is reported in the ThreadAnalyzer summary section.

This section also describes how you can use WebSphere Development Studio Client for iSeries to investigate a suspected deadlock scenario.

Troubleshooting deadlock with ThreadAnalyzer

The ThreadAnalyzer tool available via the IBM Support Assistant (refer to Appendix B, “IBM Support Assistant” on page 181) can detect whether any deadlocks are occurring. ThreadAnalyzer is designed primarily to analyze Javadump files (also known as thread dumps) from WebSphere Application Server. Because the Javadump is created at the JVM level, however, you can also use it to analyze stand-alone applications.

You can analyze thread usage at several different levels, starting with a high-level graphical view, and drilling down to a detailed tally of individual threads. If any deadlocks exist in the thread dump, ThreadAnalyzer detects and reports them. It is straightforward using ThreadAnalyzer to establish if deadlock is the cause of a hang, which is very useful in our “dining philosophers” example.

Example stand-alone deadlocked application

The example application is a classic “dining philosophers” scenario leading to *circular deadlock*. Such a deadlock involves multiple threads where each requires a resource held by another in order to progress.

The basic premise of the “dining philosophers” scenario is that a group of philosophers go to dinner. At the restaurant, there are only a limited number of knives and forks with which to eat. This means not enough knives and forks for everyone to eat at the same time. A philosopher is either eating, waiting for the knife/fork, or thinking between mouthfuls. A philosopher eating at this particular restaurant is restricted as follows:

- They can only eat if they have both the knife and fork.
- They can only pick up either the knife or a fork at any time, but not both.
- After a philosopher picks up either a knife or fork, they cannot put it down until they have eaten something.

The philosophers are independent people, therefore, each of them eats or waits or thinks for different lengths of time. Because of this, inevitably the situation eventually arises that one philosopher has the fork, while another has the knife. Both are waiting for the other to release the utensil they require. However, because of the last restriction that neither might put their utensil down until they have eaten, the philosophers cannot make any progress and are therefore deadlocked.

The deadlock in this example is usually resolved by lifting one of the restrictions, such as the restriction that a philosopher might not put the knife/fork down until they have eaten something. The philosopher instead waits for a period of time, and if they do not get the other utensil, then they put down the one they currently hold without having eaten. Such a move prevents deadlock from occurring in future. In an application this would be implemented by a thread yielding the resource it currently holds, after waiting unsuccessfully for the other resource it requires, to become available. The thread would then try again after a random period of time.

Generating the Javadump

A Javadump file is required as input to ThreadAnalyzer. This can be generated non-destructively by giving the following command in a Qshell session. You have to open a new Qshell session because the session running the JVM is not going to be responsive to input:

```
kill -QUIT process ID
```

Important: The JVM continues running after the **kill** command is issued. However, because the application cannot progress in a hung state, you can eventually terminate the JVM job manually through WRKACTJOB in order to restart the application.

The process ID is that of the hung JVM. It can be found at the top of the JVM’s job log. To display the job log, enter `WRKACTJOB`. Put an Option “5-Work with” next to the hung JVM, then choose Option “10-Display job log...”

In our case the process ID of the hung JVM is 9793 as shown in the job log in Figure 8-2.

We run the following command in a Qshell session to generate the Javadump required by ThreadAnalyzer:

```
kill -QUIT 9793
```

Refresh the job log by pressing F5. You can see messages posted in the job log as the Javadump is created, also shown in Figure 8-2.

The name and location of the Javadump file is given in the job log so that you can find it easily. In our case the Javadump file is as follows:

```
/home/rumsbya/javacore.20061004.140450.9607.txt
```

See Figure 8-2.

```
                              Display Job Log
                                                              System:    RCHAS60
Job . . :   QPOZSPWP      User . . :   RUMSBYA       Number . . . :   210932

    Java Virtual Machine is IBM Technology for Java. PID(9793)
    JVM properties were loaded from a properties file.
    Public write authority on "bughunt.jar".
    JVMDUMP006I Processing Dump Event "user", detail "" - Please Wait.
    JVMDUMP007I JVM Requesting Java Dump using
      '/home/rumsbya/javacore.20061004.140450.9607.txt'
    JVMDUMP010I Java Dump written to
      /home/rumsbya/javacore.20061004.140450.9607.txt
    JVMDUMP013I Processed Dump Event "user", detail "".



                                                                       Bottom
Press Enter to continue.

F3=Exit    F5=Refresh    F10=Display detailed messages    F12=Cancel
F16=Job menu            F24=More keys
```

Figure 8-2 The process ID of a hung JVM is found in the job log

Analyzing the Javadump with ThreadAnalyzer

The next step is to launch ThreadAnalyzer from IBM Support Assistant and determine if there are any deadlocks:

1. Ensure IBM Support Assistant is running. Refer to Appendix B, “IBM Support Assistant” on page 181 for more information about installing and configuring IBM Support Assistant.
2. Click **Tools** on the IBM Support Assistant Welcome Page.
3. Click **WebSphere Application Server 6.1** under Products.
4. Click **ThreadAnalyzer** as shown in Figure 8-3. A separate window opens with a message `Launching ThreadAnalyzer Tool....`

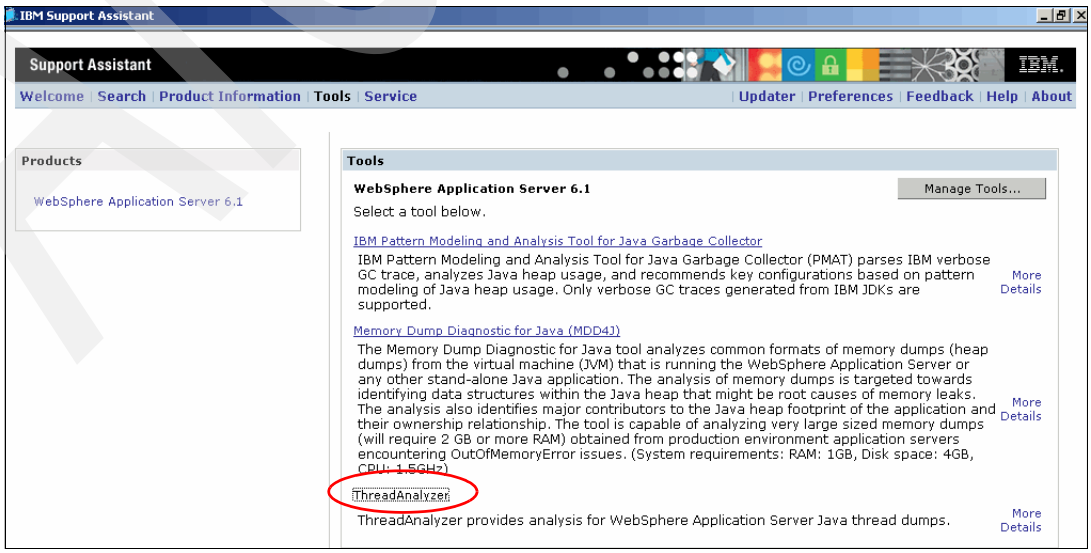


Figure 8-3 ThreadAnalyzer is accessed from the Tools menu in IBM Support Assistant

After ThreadAnalyzer starts, you have to import the Javadump file for analysis:

5. Either copy the Javadump file from the i5/OS to the PC where you are running ThreadAnalyzer, or map a drive letter to the i5/OS filesystem. We copied the Javadump file locally.
6. Select **Open Existing File** from the Thread Dumps menu. Select the Javadump file generated for this hang, as shown in Figure 8-4.

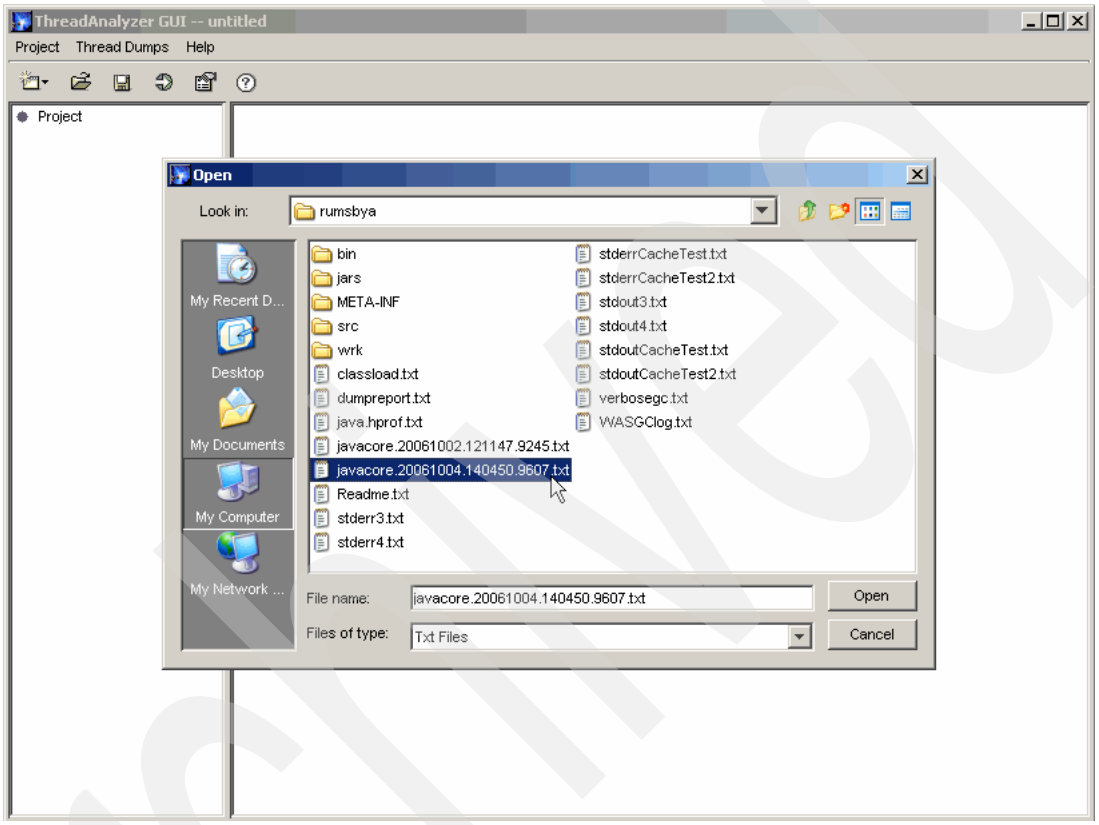


Figure 8-4 Select the Javadump to process with ThreadAnalyzer

7. Click **Open** to import the Javadump into ThreadAnalyzer
8. ThreadAnalyzer then processes the Javadump. Click **OK** on the Open Existing File(s) dialog when ThreadAnalyzer has completed processing.
9. Expand the root entry for the Javadump just processed (in this case there is only one entry) then click **Summary**.

This displays the summary information, as shown in Figure 8-5.

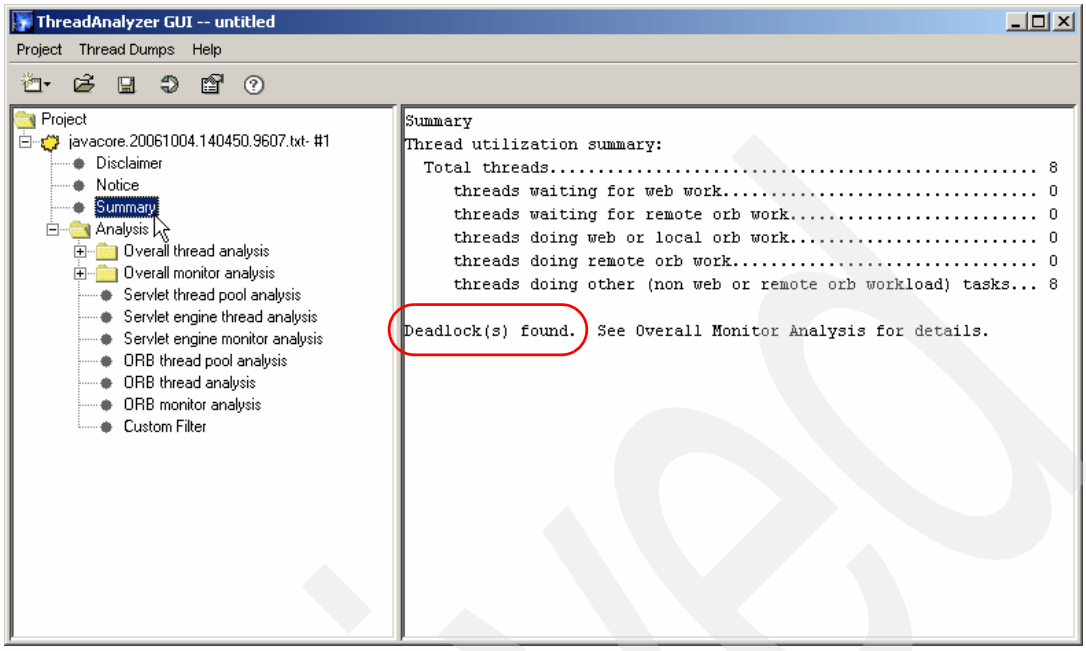


Figure 8-5 ThreadAnalyzer summary shows the deadlock

From the summary information you can see a deadlock has been detected by ThreadAnalyzer. Notice the “bang” icon next to the Javadump file name which gives visual indication deadlock has been detected.

If you click **Overall monitor analysis**, you can view the details of the deadlock, as shown in Figure 8-6.

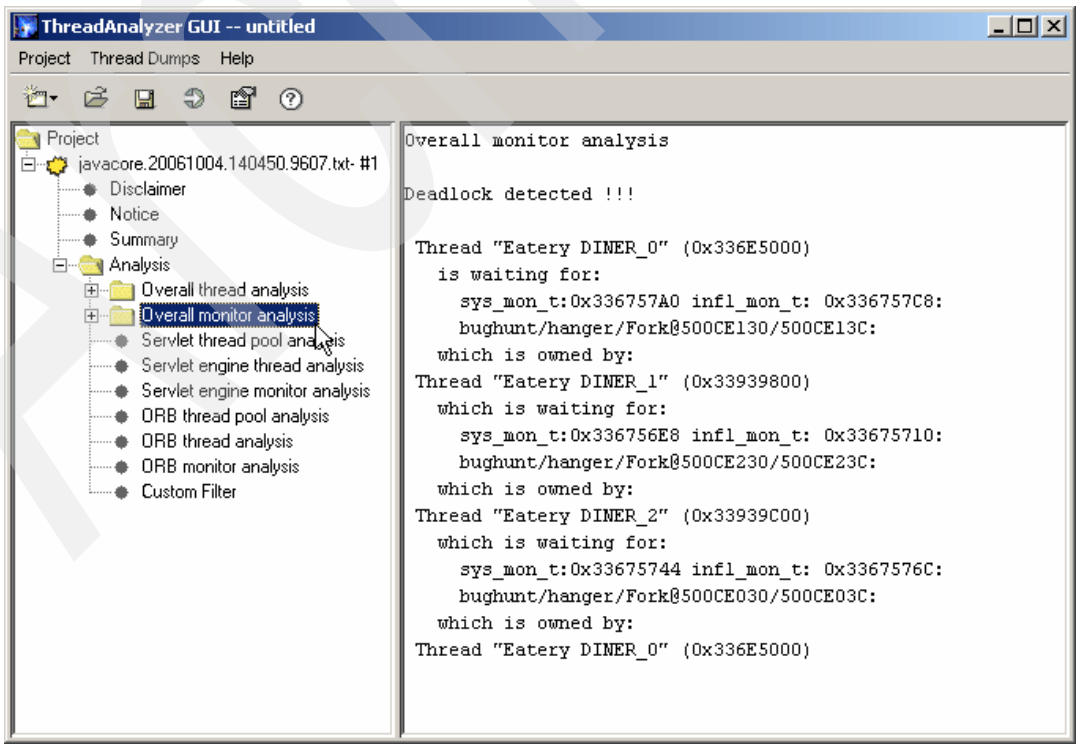


Figure 8-6 Overall monitor analysis section of ThreadAnalyzer gives more detail about the deadlock

Continuing the dining philosophers analogy, the results from ThreadAnalyzer indicate three philosophers have picked up a utensil each, but each requires another utensil (held by another of the two other philosophers) in order to continue. Because of the third restriction (that after a knife/fork has been picked up, it cannot be put down until the philosopher has eaten) none of the philosophers can make any progress and therefore a circular deadlock has been reached.

In this case because the deadlocked threads are application threads, the application would have to be debugged further to determine the cause. Refer to “Analyzing deadlocks in WebSphere Development Studio Client for iSeries” on page 104 for guidance on troubleshooting an application using WebSphere Development Studio Client for iSeries.

Using ThreadAnalyzer, you can view the call stacks of application threads. This is useful when troubleshooting a hang due to deadlock. You can establish which method the thread was executing at the time of the hang, narrowing the scope of where the problem could lie. This also clarifies if the hang is likely an application issue or (less likely) a problem with one of the Java shared class libraries.

Use ThreadAnalyzer to look at the Javadump from the dining philosopher’s example again to see which methods were being executed when the deadlock occurred.

1. If ThreadAnalyzer is not currently running, start ThreadAnalyzer and load the Javadump from the hang into ThreadAnalyzer. Refer to “Analyzing the Javadump with ThreadAnalyzer” on page 94 for more information.
2. Select the Javadump file which was generated from the hang and click **Open**.
3. Expand the **Overall thread analysis** subfolder of the report:
   a. Expand the root entry for this Javadump
   b. Expand the **Analysis** subfolder
   c. Expand the **Overall thread analysis** subfolder. You can see a summary of the methods and how many times each method is found to be at the top of a thread’s stack, as shown in Figure 8-7.

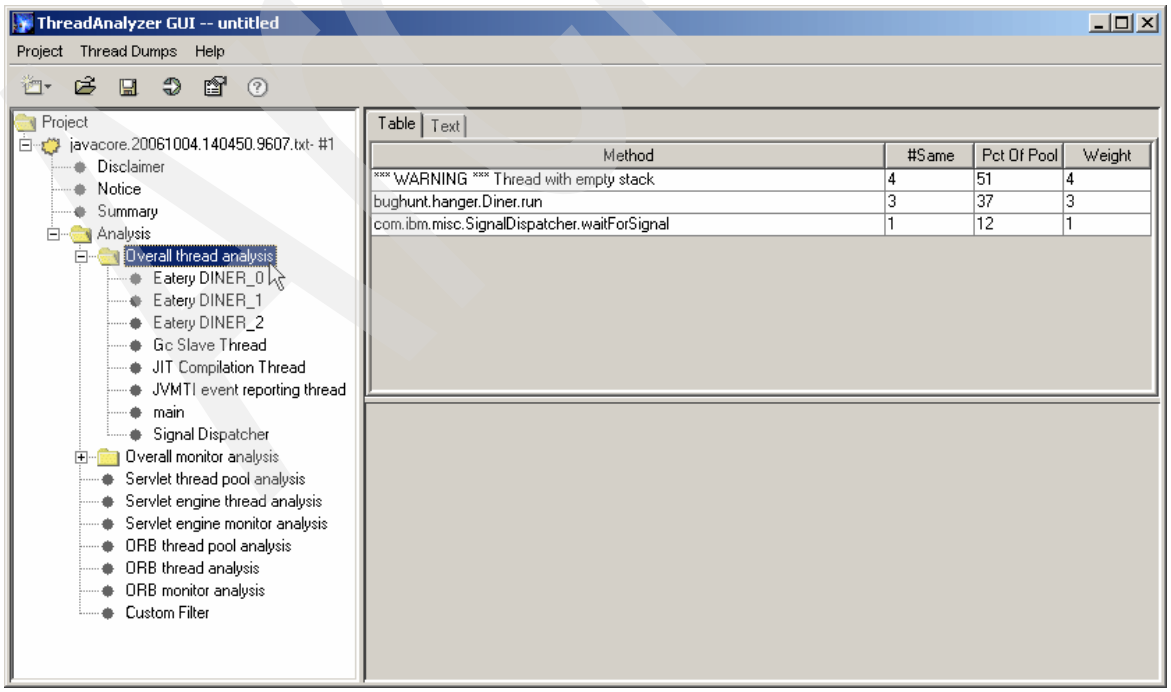


Figure 8-7 The overall thread analysis shows the occurrences of each method among the threads

Note: You might notice ThreadAnalyzer occasionally reports a thread has an empty stack. This can occur if the Javadump supplied to ThreadAnalyzer does not contain the stack trace for a thread. This is not an issue with ThreadAnalyzer.

Investigate the stack trace of the hung threads. Click the **Eatery DINER_0** thread. The thread information, including the call stack trace, is displayed, as shown in Figure 8-8.

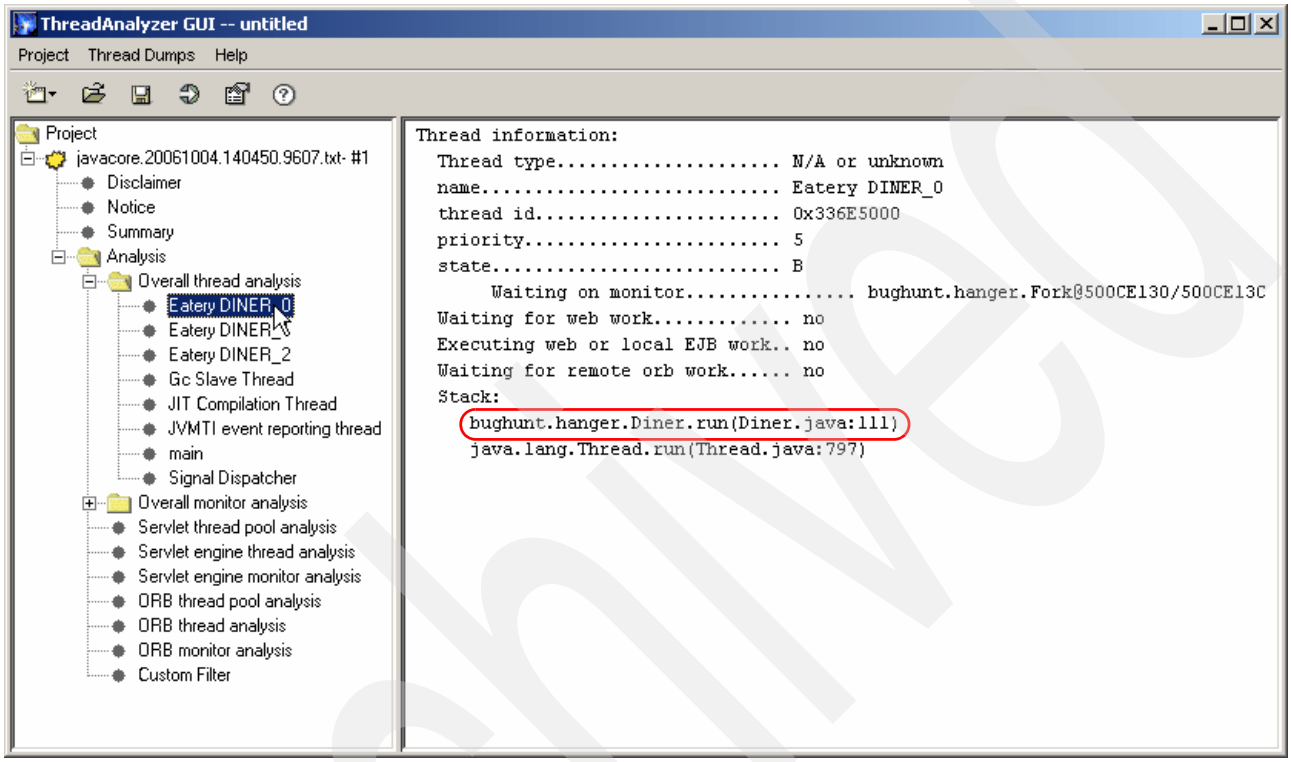


Figure 8-8 Viewing the stack trace of a deadlocked thread in ThreadAnalyzer

Notice the stack trace reported in the thread information. This shows that the run() method was active when the hang occurred. Therefore, for further troubleshooting you must investigate the code in the run() method for logic errors that could lead to a hang. 8.6, “Performance issues” on page 127 shows you how to use WebSphere Development Studio Client for iSeries to profile the application.

Example WebSphere Application Server-based application

The ThreadAnalyzer tool is primarily designed for analyzing thread dumps (Javadumps) captured in WebSphere Application Server environments, where there can be hundreds of threads in an application. For such applications, manual analysis of thread dumps is time-consuming and unreliable in pinpointing the cause of a performance or hang-related problem.

Note: This section applies only to WebSphere Application Server V6.1 and later, because support for IBM Technology for JVM was introduced in this version.

Deadlock situations are reported by ThreadAnalyzer, similar to stand-alone applications, as described in “Analyzing the Javadump with ThreadAnalyzer” on page 94.

The remainder of this section is focused on issues where an application running under WebSphere Application Server exhibits poor performance or hangs without a deadlock.

ThreadAnalyzer includes a custom filter feature that allows you to reduce the amount of thread data that is displayed in the thread analysis view. You can also work with multiple thread dumps simultaneously in a comparative analysis.

We used ThreadAnalyzer to investigate a performance issue in a WebSphere Application Server environment. A Web application was giving very poor performance under WebSphere Application Server. We captured several thread dumps during the application lifetime so that we could use the ThreadAnalyzer's comparative analysis feature to determine which threads were making progress and what methods they were calling.

Note: Using ThreadAnalyzer to troubleshoot WebSphere Application Server environments requires you to understand the architecture of the application and be able to distinguish between core WebSphere Application Server methods and methods which belong to the application.

The procedure used to troubleshoot this particular application is as follows:

1. Start the application and recreate the conditions which cause poor performance or monitor for signs of poor performance.
2. Capture multiple thread dumps (Javadumps) when the problem surfaces.

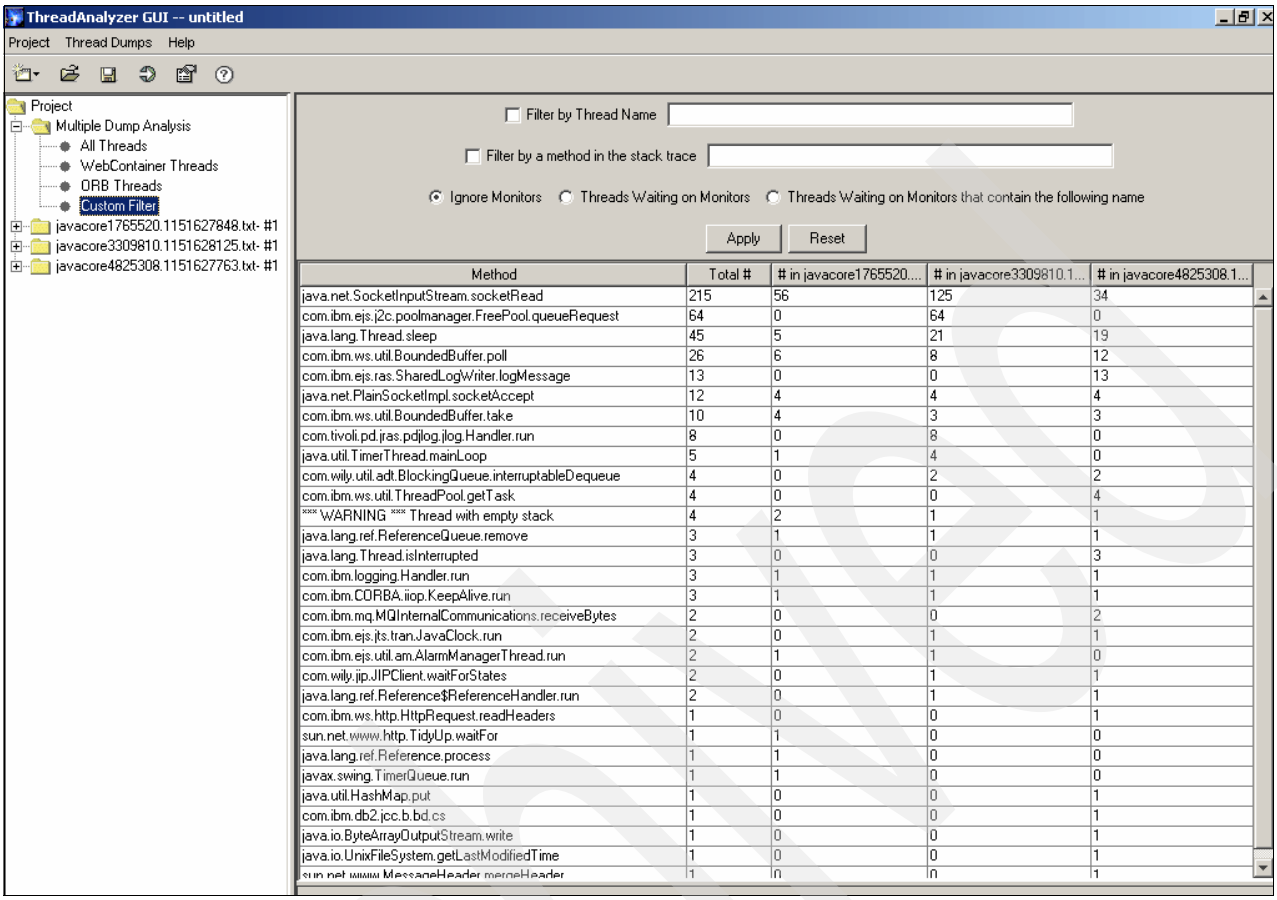
   Refer to "Generating the Javadump" on page 93 for information about how to capture a Javadump. Because the application we are analyzing is running under WebSphere Application Server, the process ID of the WebSphere Application Server JVM has to be passed to the `kill -QUIT` command.

   You can distinguish the WebSphere Application Server JVM from others that might be running, because it will be running in the QWAS61 subsystem.

   There is no "magic number" of thread dumps to capture, it varies by application and how soon the problem surfaces.
3. Either copy the Javadump files from the i5/OS to the PC where you are running ThreadAnalyzer, or map a drive letter to the i5/OS filesystem. We copied the Javadump files locally.
4. Launch ThreadAnalyzer from the IBM Support Assistant. Refer to "Analyzing the Javadump with ThreadAnalyzer" on page 94 for information about how to start the ThreadAnalyzer.
5. Open the thread dumps to be analyzed. Because we are comparing multiple thread dumps, use Ctrl-click to select multiple thread dumps to analyze. Refer to "Analyzing the Javadump with ThreadAnalyzer" on page 94 for information about how to open thread dumps.
6. Expand **Multiple Dump Analysis** under the Project folder, then select **Custom Filter**.

   The resulting view sorts each method by the total number of threads which have that method on the top of their stack, as shown in Figure 8-9.

See Figure 8-9.



Method	Total #	# in javacore1765520....	# in javacore3309810.1...	# in javacore4825308.1...
java.net.SocketInputStream.socketRead	215	56	125	34
com.ibm.ejs.j2c.poolmanager.FreePool.queueRequest	64	0	64	0
java.lang.Thread.sleep	45	5	21	19
com.ibm.ws.util.BoundedBuffer.poll	26	6	8	12
com.ibm.ejs.ras.SharedLogWriter.logMessage	13	0	0	13
java.net.PlainSocketImpl.socketAccept	12	4	4	4
com.ibm.ws.util.BoundedBuffer.take	10	4	3	3
com.tivoli.pd.jras.pdjlog.jlog.Handler.run	8	0	8	0
java.util.TimerThread.mainLoop	5	1	4	0
com.wily.util.adt.BlockingQueue.interruptableDequeue	4	0	2	2
com.ibm.ws.util.ThreadPool.getTask	4	0	0	4
*** WARNING *** Thread with empty stack	4	2	1	1
java.lang.ref.ReferenceQueue.remove	3	1	1	1
java.lang.Thread.isInterrupted	3	0	0	3
com.ibm.logging.Handler.run	3	1	1	1
com.ibm.CORBA.iiop.KeepAlive.run	3	1	1	1
com.ibm.mq.MQInternalCommunications.receiveBytes	2	0	0	2
com.ibm.ejs.jts.tran.JavaClock.run	2	0	1	1
com.ibm.ejs.util.am.AlarmManagerThread.run	2	1	1	0
com.wily.jip.JIPClient.waitForStates	2	0	1	1
java.lang.ref.Reference$ReferenceHandler.run	2	0	1	1
com.ibm.ws.http.HttpRequest.readHeaders	1	0	0	1
sun.net.www.http.TidyUp.waitFor	1	1	0	0
java.lang.ref.Reference.process	1	1	0	0
javax.swing.TimerQueue.run	1	1	0	0
java.util.HashMap.put	1	0	0	1
com.ibm.db2.jcc.b.bd.cs	1	0	0	1
java.io.ByteArrayOutputStream.write	1	0	0	1
java.io.UnixFileSystem.getLastModifiedTime	1	0	0	1
sun.net.www.MessageHeader.mergeHeader	1	0	0	1

Figure 8-9 Multiple Dump Analysis aggregates data from multiple Javadump files

The number of occurrences of threads with each method at the top of the stack, is reported in a separate column for each Javadump also. Comparing these figures gives an easy indication if the application has made any progress in the time between capturing each Javadump. For larger applications, there can be hundreds of threads, and correspondingly, hundreds of methods listed in this view.

Many of the methods are executed by core WebSphere Application Server threads, because the application we are analyzing runs in a WebSphere Application Server environment. It is easier to work with just the application threads that are involved in the problem and discount the WebSphere Application Server threads.

This particular application was composed mostly of Servlets, therefore we want to focus on the threads that run the servlet code, that is *Servlet handler* threads.

7. In the Filter by Thread Name field enter `Servlet` then click **Apply**. The number of methods displayed is reduced, as shown in Figure 8-10.

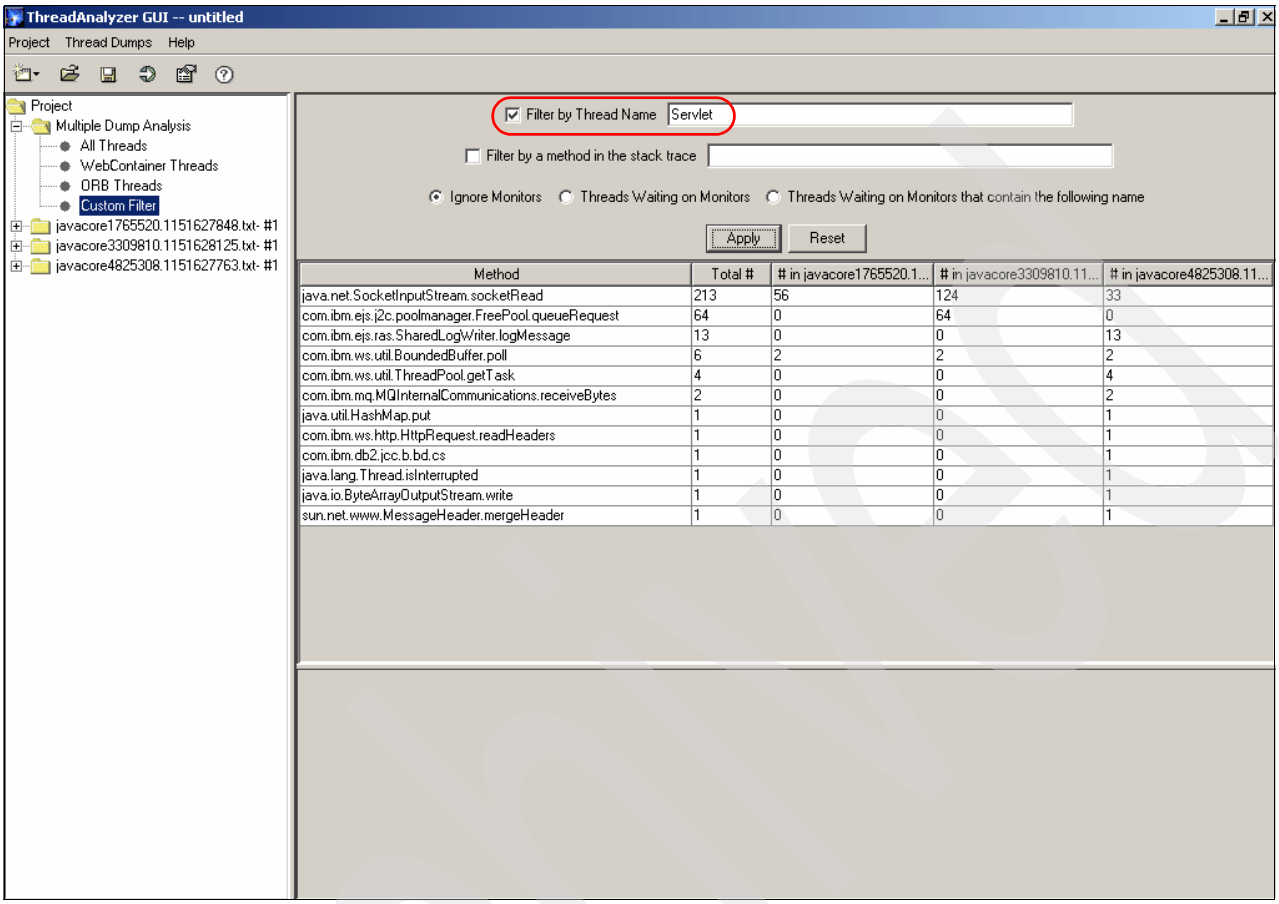


Figure 8-10 ThreadAnalyzer's Custom Filter feature lets you focus on methods related to the application

You then have to analyze the methods further. Start with the top method listed in the summary because it was the most common method being executed when the thread dump was captured.

8. Double-click the top method, in this case **java.net.SocketInputStream.socketRead()**.

   Another view opens showing a tree structure under which the threads that have this method at the top of their stack, are displayed.

9. Click the first thread dump file name to show the threads that have this method at the top of the stack. Notice the thread's call stack is displayed on the right, as shown in Figure 8-11.

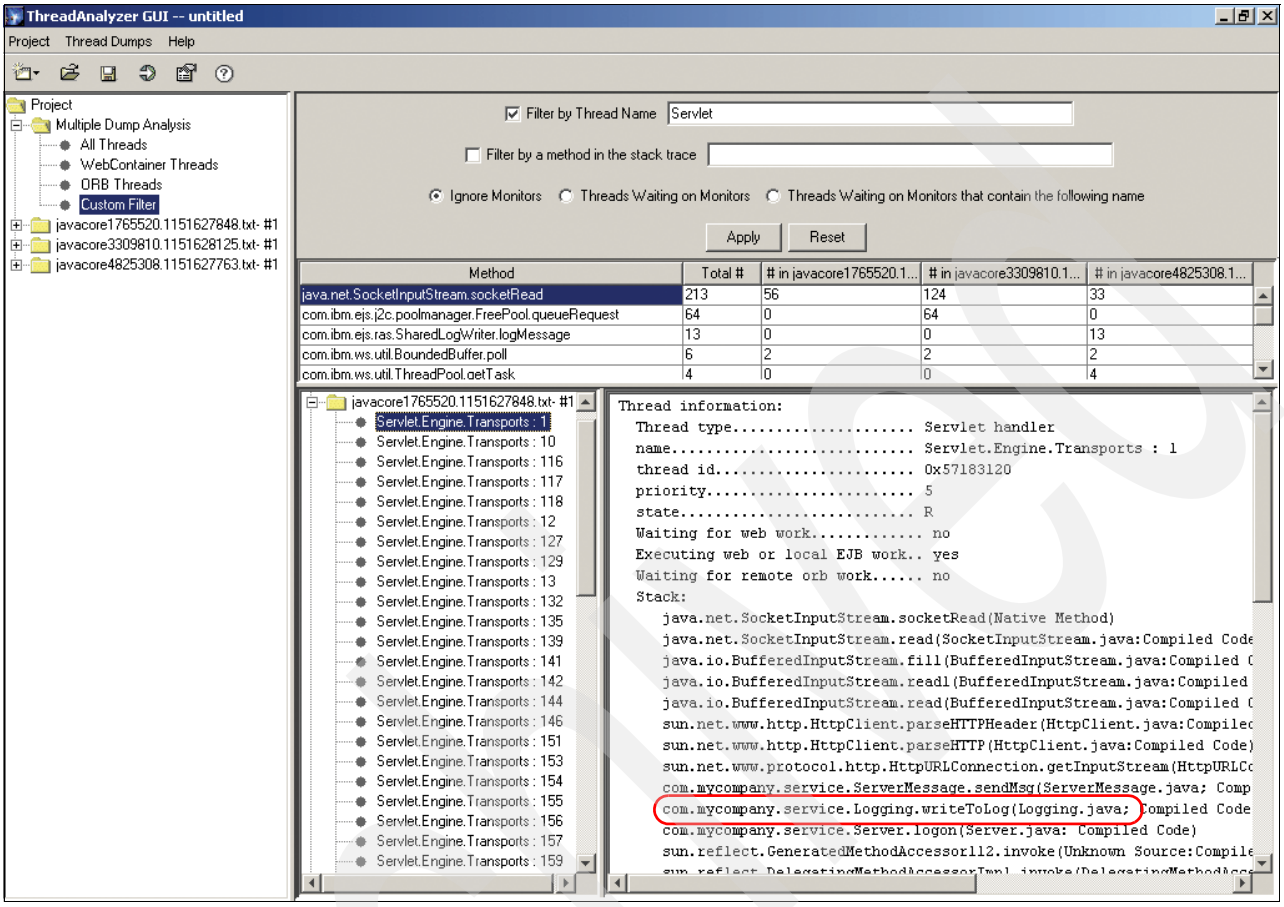


Figure 8-11 Reviewing a thread's call stack in ThreadAnalyzer

The next stage is to analyze the thread call stacks that have this method at the top. You are looking for a clue or common application methods which opens in many threads' stacks. If you find such a method, it warrants further investigation.

10. We did this by pressing the **down arrow** key and watching the stack in the right pane, to see if and how it changed between each thread. We noticed that the method writeToLog() was displayed in every thread's call stack.

    From the package name, we knew this was not a method from one of the classes supplied in the standard Java Class Libraries. Therefore, we decided to investigate this method further.

11. We further filter the thread data displayed by specifying writeToLog in the Filter by a method in the stack trace field, then clicking **Apply**, as shown in Figure 8-12.

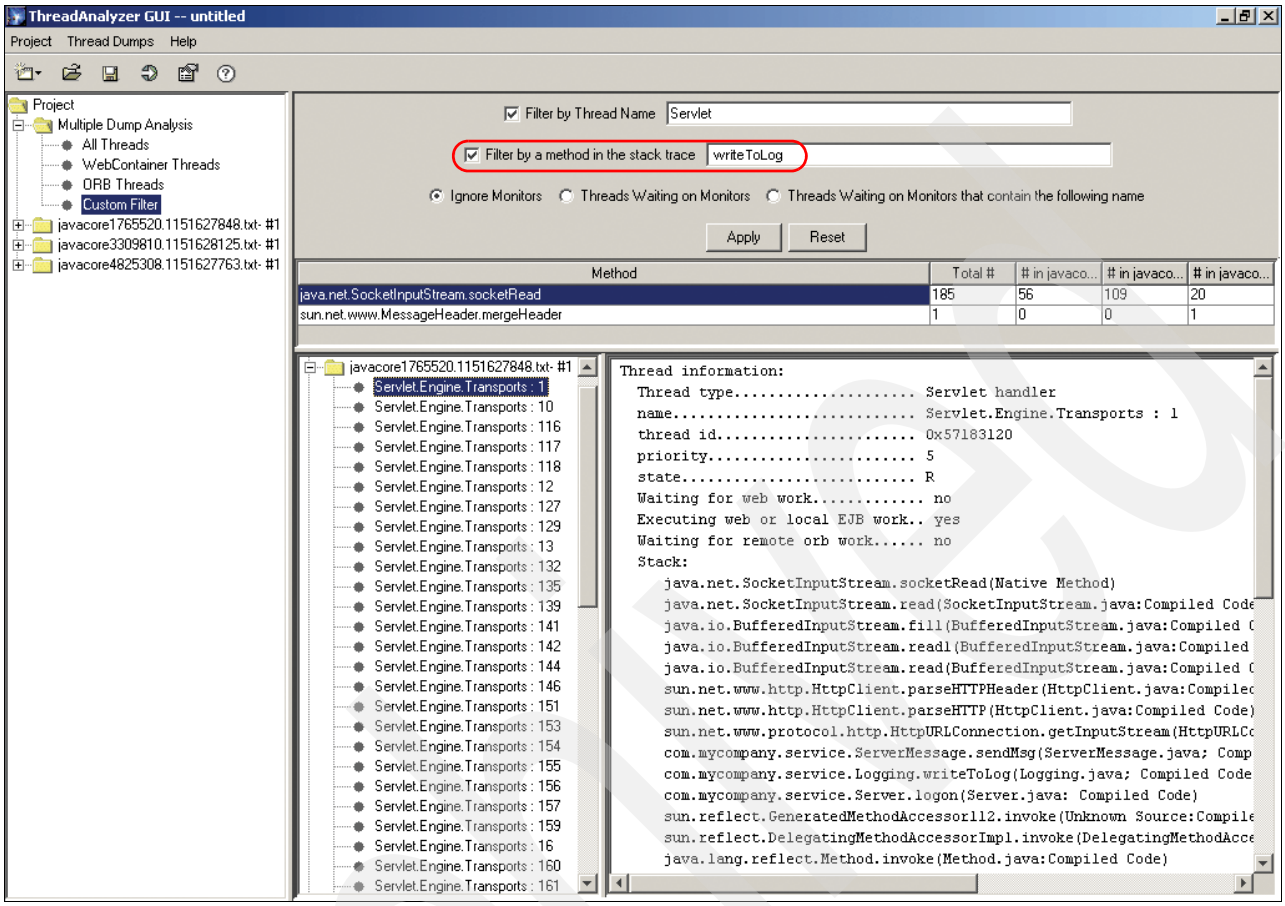


Figure 8-12 Finding occurrences of a method using ThreadAnalyzer's custom filter feature

From Figure 8-12, you can see that it is only threads with the java.net.SocketInputStream.socketRead() method at the top of the stack, which also have called the writeToLog() method.

Remember that the java.net.SocketInputStream.socketRead() method was the most common "top of stack" method in the application. The implication is that most of the threads also spend time in the writeToLog() method.

12. This insight prompts you to ask the following questions:
    - What is the function of the writeToLog() method?
    - Why is it called by so many threads?
    - Why does it take so long to return?

13. These are questions that you resolve by going to the source code for the writeToLog() method.

    After reviewing the source code, you discover this method is used for logging to a custom log file. The method makes a synchronous call to a remote JVM to do the logging. The inefficient operation of this method, combined with the sheer number of times it is called, causes noticeable perturbation to the application.

    The issue is resolved in this instance by rewriting the writeToLog() method to use asynchronous logging by the current JVM rather than using expensive remote, synchronous calls.

Analyzing deadlocks in WebSphere Development Studio Client for iSeries

A thread deadlock occurs when two independent threads of control are blocked, each waiting for the other to take some action. Two threads waiting for each other can bring an application, or one section of the application, to a complete halt.

A thread contention, or race conditions, occurs when a thread is waiting for a lock or resource that another thread holds. Programmers often add synchronization mechanisms to avoid these contentions, but it is possible that the synchronization itself can lead to deadlocks, if done incorrectly. In this section you see how to identify a deadlock situation between threads.

Restriction: At the time of this writing, WebSphere Development Studio Client for iSeries version 6.0.1 has not supported J9 for remote debugging. However the approach presented in this section is applicable to IBM Technology for JVM when this is supported for remote debugging.

Perform the following steps:

1. Start WebSphere Development Studio Client for iSeries Advanced Edition on your workstation.
2. Open the Java perspective.
3. Locate the Java application project and source files for your application.
4. Right-click your mouse on the class (which must have a main() method) and select **Profile** → **Java Application**.
5. At the properties window select the **Profiling** tab.
6. Select the **Thread Analysis** option and click **OK** as shown in Figure 8-13.

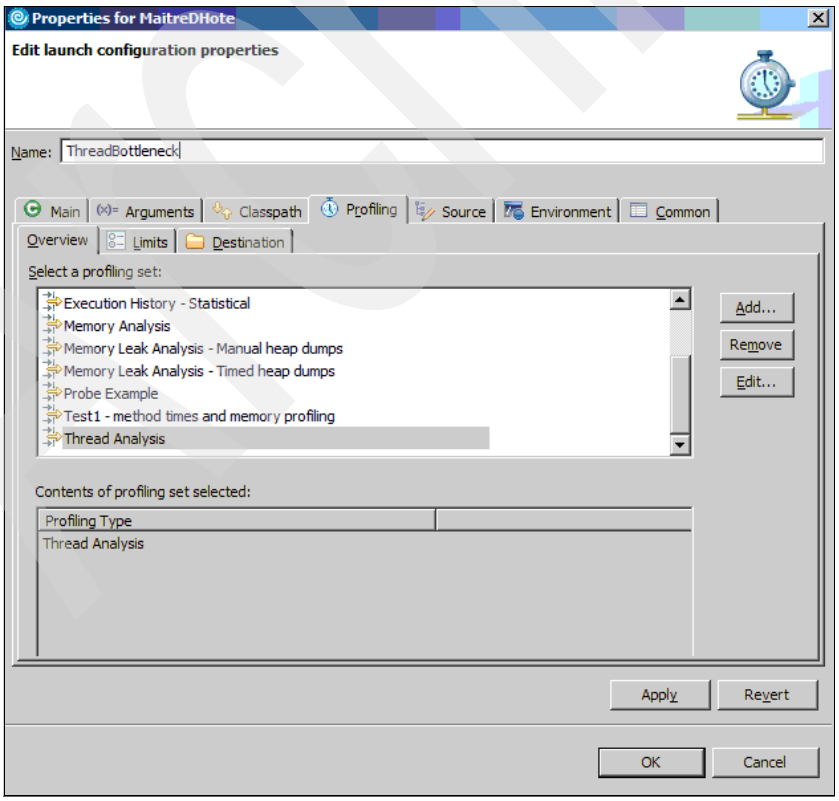


Figure 8-13 Thread analysis selection

7. If you do not have the profiling and logging perspective open you see a “confirm perspective switch?” window open. If so, click **Yes**.
8. You must be in the profiling and logging perspective and see your application status as monitoring. If the console does not open, go to the toolbar and select **Window** → **Show View** → **Console**. This is going to display any output data.
9. You have to run the application to completion.
10. After the application completes you must see your application status as terminated. Expand the application link and you must see a Thread Analysis option. Right mouse click the Thread Analysis option and select **Open With** → **Thread view**.
11. The thread view is displayed, as shown in Figure 8-14. If there are thread deadlocks detected, then they are indicated.

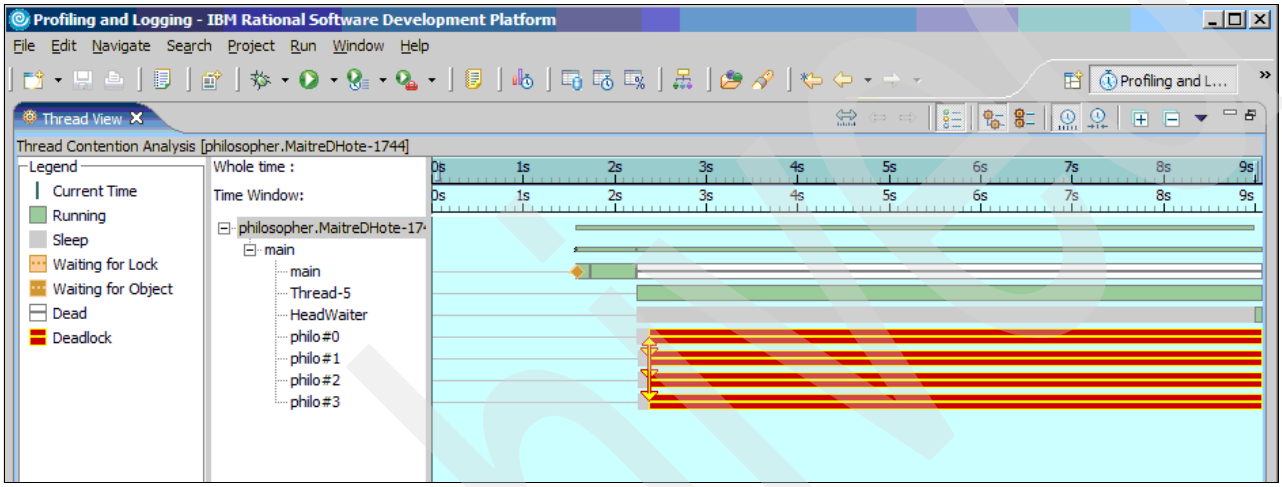


Figure 8-14 Default thread view example

The vertical arrows between threads are of interest. An arrow indicates that one thread (the thread in which the arrow originates) is waiting for another thread (the thread to which the arrow is pointing) to release a lock. A double arrow indicates that two threads are in a deadlocked state, both waiting for the other to release a lock.

12. The initial view may be difficult to work with. Click **Switch to Compressed Time** button to view a more user friendly representation of the thread activity, as shown in Figure 8-15.

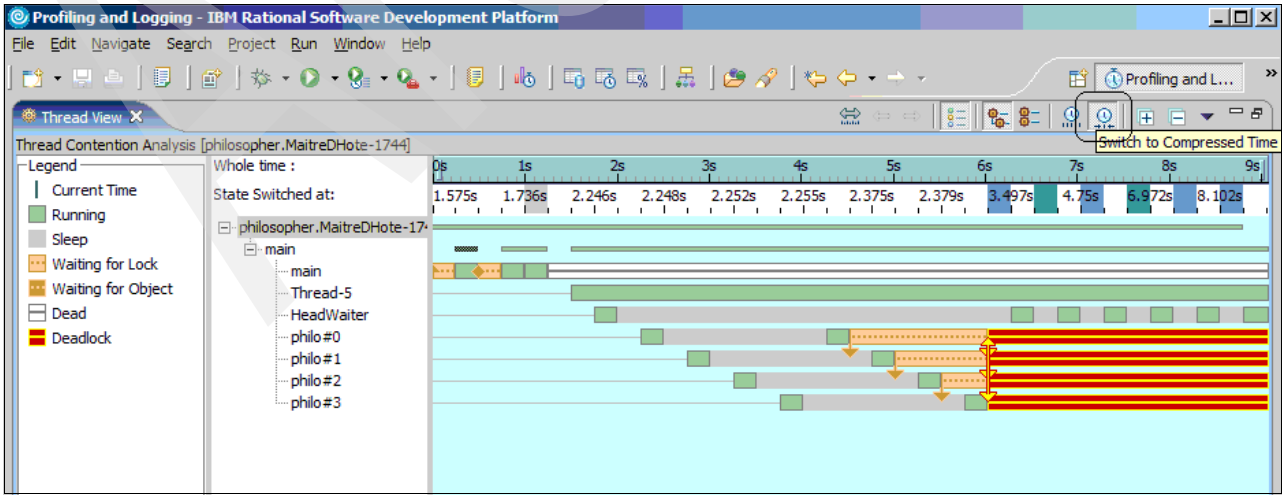


Figure 8-15 Alternative thread view

Soon after the program starts, four philo#x threads are created, one after the other. Each thread runs for a short while and then makes a request for a lock. The requests are not successful. When the first three lock requests fail, the threads enter a Waiting for Lock state. When the lock request for the fourth thread fails, deadlock occurs, and the display indicates that all four threads are in a Deadlocked state.

Notice the vertical arrows between the threads. An arrow indicates that one thread (where the arrow originates) is waiting for another thread (where the arrow points) to release a lock. A double arrow indicates that two threads are in a deadlocked state, both waiting for the other to release a lock.

In this example the philo#x threads are waiting for a lock from other philo#x threads that are also waiting for a lock. There is now a thread deadlock and the program cannot continue.

To resolve this issue you have to find those method calls and objects that are involved. You can use the UML2 Sequence Diagram (the Object Interactions and Thread Interactions views) and the Call Stack View, in addition to the Thread View to assist you.

13. In the Thread View, find the first thread that enters a “Deadlock” state. Pause your cursor over the “Deadlock” segment. You must see the lock (Fork.<id number>) and the locking thread (Locking Thread.<name>). For example, as follows:

```
Fork.8008
Locking Thread.philo#1
```

See Figure 8-16.

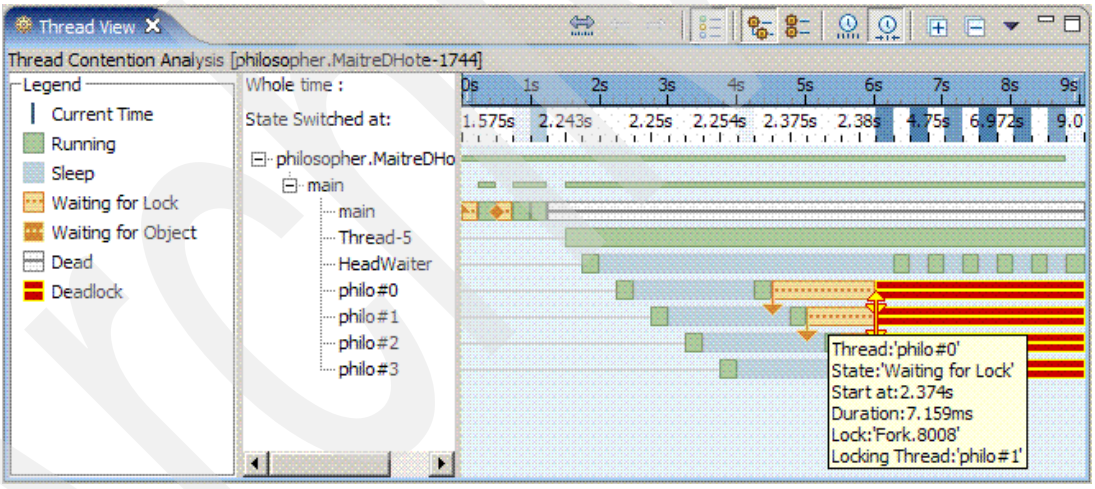


Figure 8-16 Identify the thread lock source

14. Right-click the Profiling resource for the run and select **Open With** → **UML2 Object Interactions**, as shown in Figure 8-17.

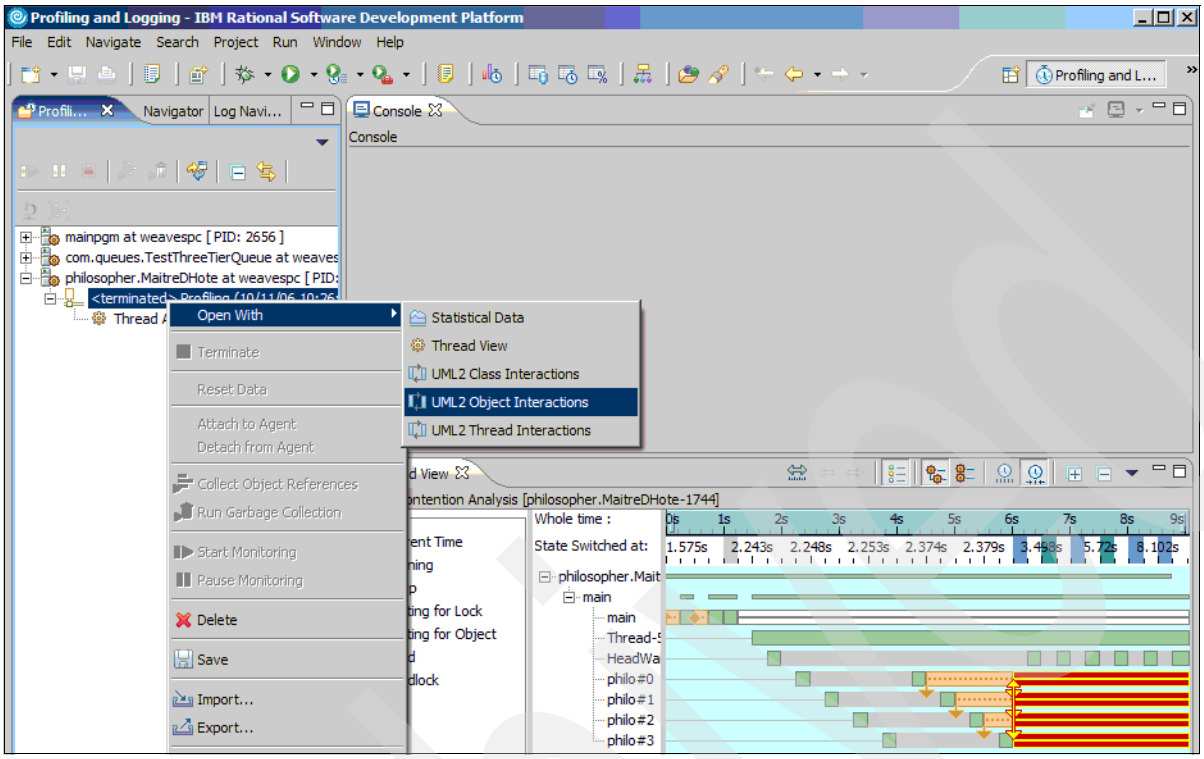


Figure 8-17 Opening the UML2 Object Interactions view

15. The UML2 Trace Interactions view opens, showing the object Interactions.
16. Scroll across the view to find the appropriate Fork.<id number> icon and select it, as shown in Figure 8-18.
17. Scroll down the view to find the Locking Thread.<thread id> interaction.

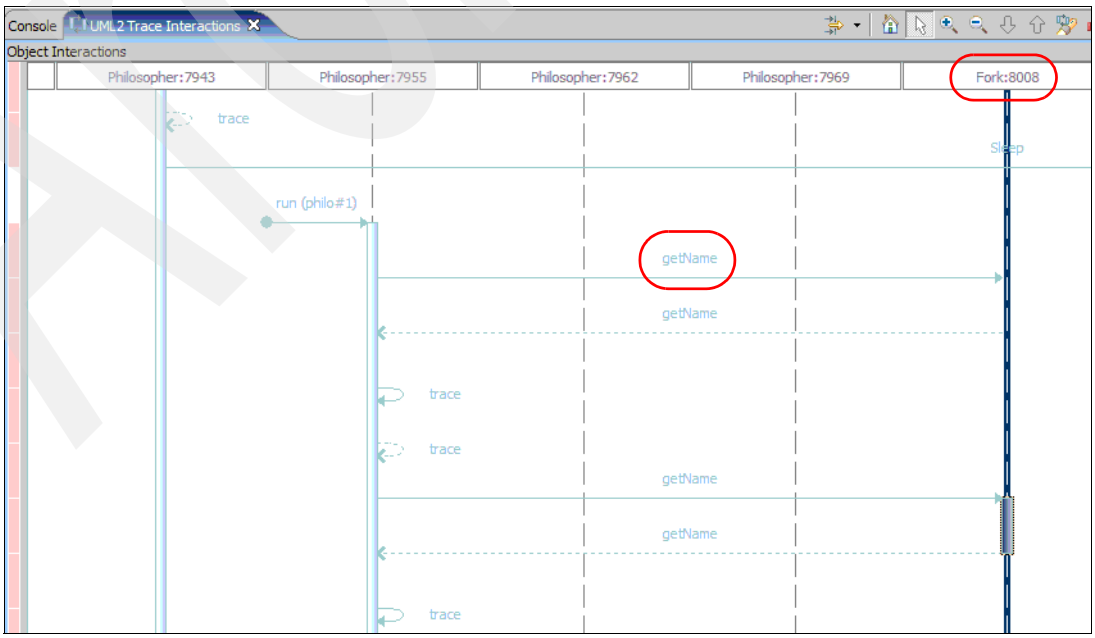


Figure 8-18 Locate the object interaction

18. Click the method (**getName()** in Figure 8-18) and the arrow must be highlighted as shown in Figure 8-19. You must also see a cursor in the thread view for this point in time.

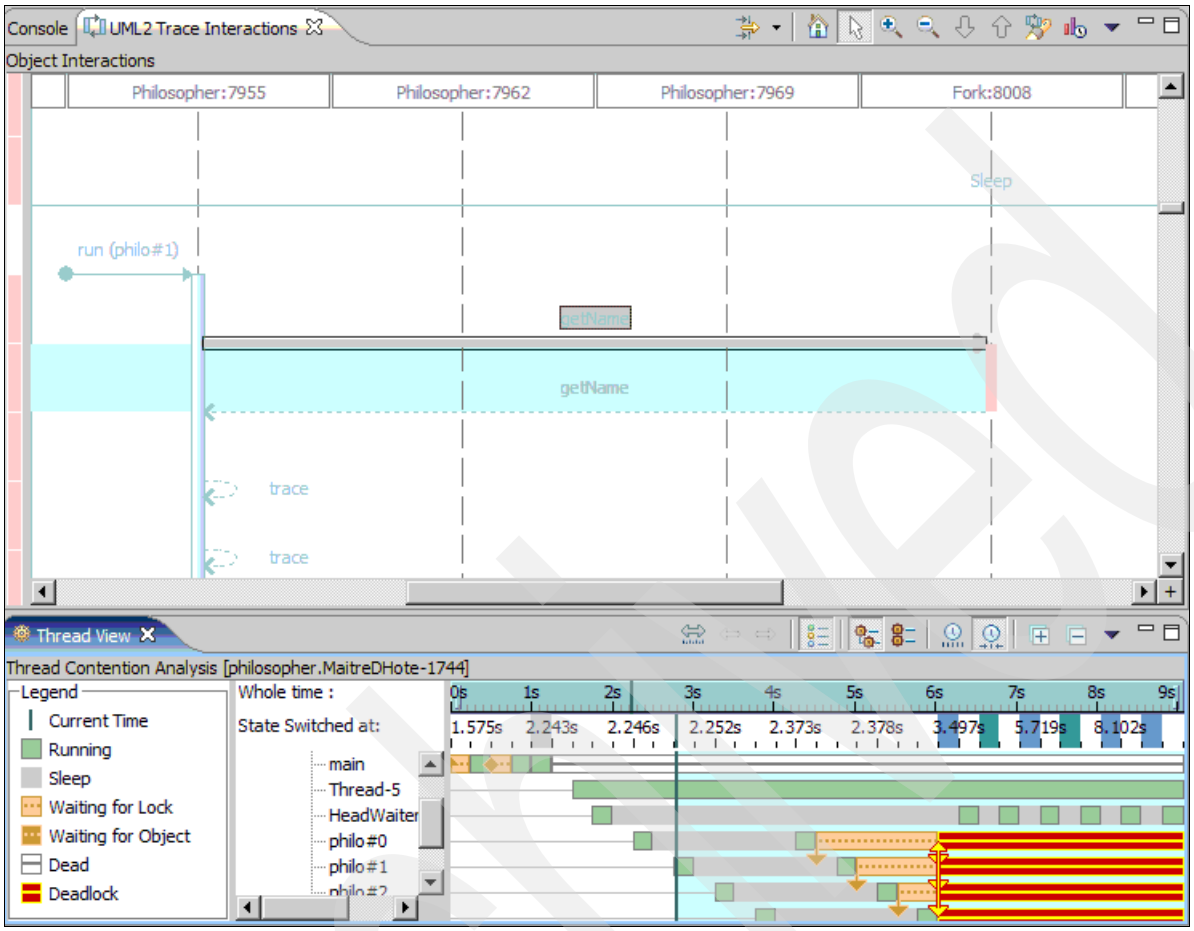


Figure 8-19 View thread status at program call time

19. In this example you can see that thread philo#1 has started to run and has seized the Fork object (Fork:8008 in this example). In the Thread view you can see that there are no threads waiting for locks or any deadlocks.

20. In the UML2 trace interactions view scroll down to find the request for the same resource, that originated in a different thread (the getName() method in this example).

21. Click the **getName()** method and the arrow must be highlighted as shown in Figure 8-20. You must also see a cursor in the thread view for this point in time.

    In this example you can see that thread philo#3 is starting to run and several other threads have started to run but then stopped to wait for a lock. This indicates a deadlock is going to occur shortly. The next task is to determine the method that is causing the problem.

See Figure 8-20.

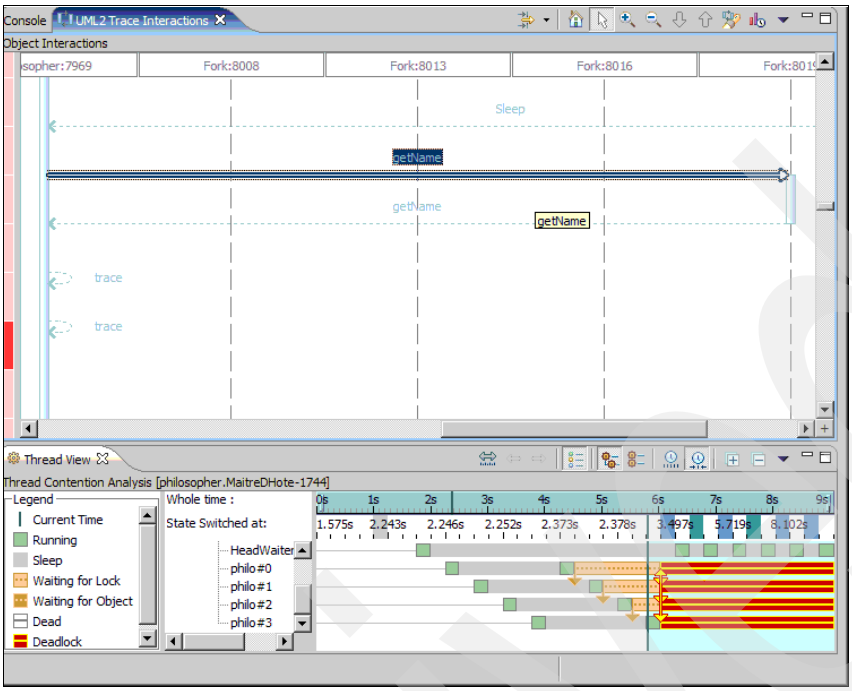


Figure 8-20 Thread locking example

22. Right-click the Profiling resource for the run and select **Open With → UML2 Thread Interactions**.

23. In the Thread View, click the menu drop-down button and select **Open Call Stack View** as shown in Figure 8-21.

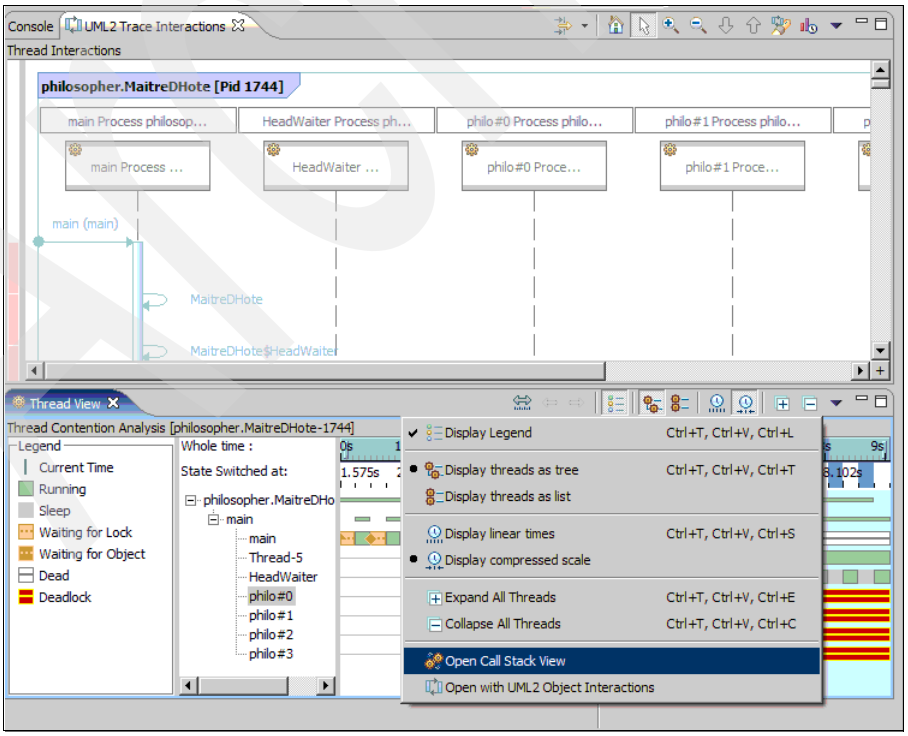


Figure 8-21 View the thread interactions and call stack

24. Locate the first deadlocked thread graph in the thread view.

25. Scroll through the information for the thread and double-click the last method that the thread ran successfully. The call stack view displays all the calls on the stack at that time as shown in Figure 8-22.

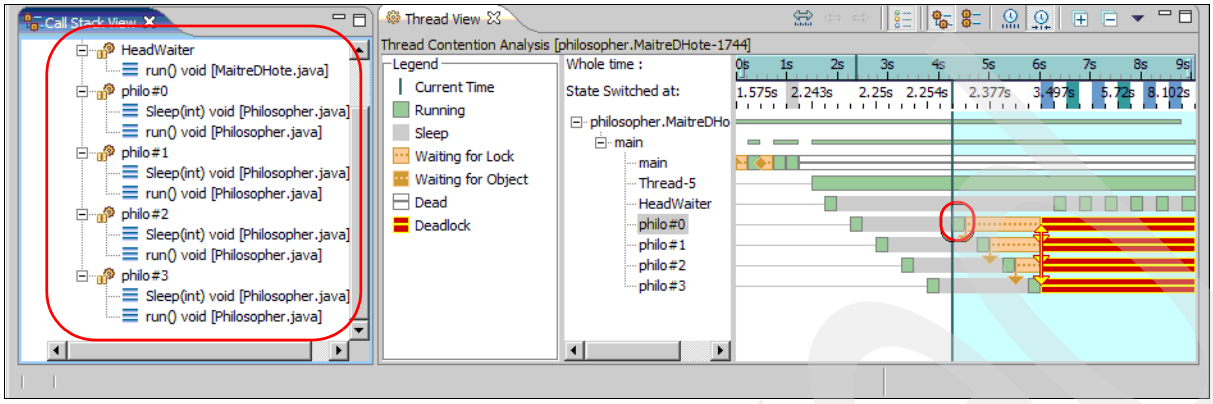


Figure 8-22 View the call stack for the active thread

26. At this time you must analyze the source code where the deadlock has occurred (based on the call stack). Select the **Open source** option to open source code as shown in Figure 8-23.

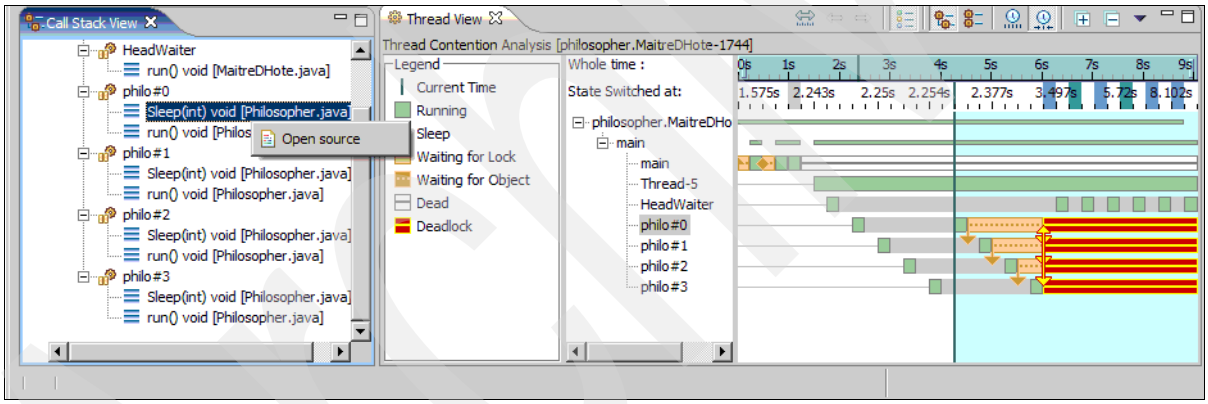


Figure 8-23 Open the source for the method identified with the deadlock

Troubleshooting memory constraints

You saw that the JVM might crash if there is insufficient memory. Before such a situation occurs, the application performance starts to degrade as available memory becomes more scarce. Classic JVM memory might start being paged out to disk and thrashing might occur whereby the system spends most of its time swapping pages between memory and disk. This leaves less processing time available for the application to progress. Such a situation shows itself through worsening response times and gives the impression that the application has hung.

With IBM Technology for JVM, however, depending on how much memory is installed, the JVM is likely to run out of memory before i5/OS has to start paging. This is because IBM Technology for JVM can use only approximately 3 GB of memory, which might be containable in main memory without the requirement to page. Therefore, you cannot necessarily rely on paging to occur to let you know that the JVM is having a memory issue when using IBM Technology for JVM.

You can quickly check the WRKSYSSTS output. If you are seeing page faults in the pool used by the JVM, then there is clearly a memory usage problem. If the cause of the hang has not been found by this stage, you must check for a memory leak by following the guidance provided in 8.5, “Investigating a suspected memory leak” on page 111.

8.5 Investigating a suspected memory leak

Java memory leak issues are probably the most frustrating and time consuming problems to troubleshoot. If you suspect a memory leak in your Java application you are likely experiencing one or more of the following symptoms:

- Abnormal end/crash due to an OutOfMemory condition
- High page faults (observed via WRKSYSSTS or performance data)
- Slow application response times

This section describes how to approach the solving of such problems. This sections begins by offering some simple suggestions and moves toward some more advanced analysis tools and techniques.

8.5.1 Quick fix

As already mentioned that memory leaks are difficult issues to resolve, the chances of finding a “quick fix” to the problem is probably not very good. However, it might be possible to find a circumvention that would allow a production application to continue running while you investigate the memory leak. Consider the following scenarios:

- Did this apparent memory leak surface after an existing application was updated?

  Potential circumvention:

  - If possible, deploy the previous version of the application.
  - Discuss the problem with the application developers that made the changes.

- Did the problem surface after i5/OS PTFs were installed?

  Potential circumvention:

  - Find out what PTFs were installed and contact IBM support to see if any problems have been reported with those PTFs.
  - Try to verify all properties or settings required by your application. If you have any sort of configuration files stored in an i5/OS product directory, these might have been overlaid by the installation of PTFs.

8.5.2 Memory leaks and the heap

The IBM Technology for JVM maintains two separate areas of memory. Together they represent the total memory footprint. These are the Java heap and *native heap*. The Java heap is the one that contains all the Java objects and is managed by the garbage collector. In the context of Java, most references to “heap” are referring to Java heap. The minimum and maximum size of the heap are configured using the -Xms and -Xmx settings.

The contents of the native heap are more stable than Java heap and are not subject to garbage collection. The native heap is typically used when application JNI code does a malloc(). Space for code that is optimized by the JIT compiler is also allocated from the native heap. The size of the native heap is influenced by the -Xmx setting, because both heaps must fit within a 32-bit address space, you can increase the size of the native heap by decreasing the amount used by the Java heap, using the -Xmx setting.

A memory leak is much more likely to occur in the garbage-collected Java heap, therefore, the remaining sections focus on the Java heap. For more information about native heap usage on i5/OS we suggest you review Chapter 17 in the IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide on the Web at:

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

8.5.3 Collecting heap information

If none of the quick fix recommendations helped resolve your memory leak, then you have to investigate what types of objects are accumulating in the Java heap. This is done by generating heap dumps for analysis. The following instructions and sample commands describe how the heap dumps are generated. In 8.5.4, “Using the Memory Dump Diagnostic for Java (MDD4J) tool” on page 114 we describe a tool that is used to automate analysis of heap dumps.

Triggered by an OutOfMemoryError

If you are familiar with the Classic JVM on i5/OS you probably are not accustomed to seeing an OutOfMemoryError very often. The reason for this is that the default setting for maximum heap size is essentially *NOMAX. The default maximum heap size for IBM Technology for JVM is 2 GB. Because the default setting is defined and users of IBM Technology for JVM are likely to tune the maximum heap size, the odds of seeing an OutOfMemoryError increases. When the VM reaches an out of memory condition a heap dump is generated. As you can see in Figure 8-24, the location of the dump file is logged to stderr.

```
JVMDUMP006I Processing Dump Event "uncaught", detail "java/lang/OutOfMemoryError" - Please
Wait.
JVMDUMP007I JVM Requesting Snap Dump using
'/home/pauls/java/Snap0001.20061006.141801.9909.trc'
JVMDUMP010I Snap Dump written to /home/pauls/java/Snap0001.20061006.141801.9909.trc
JVMDUMP007I JVM Requesting Heap Dump using
'/home/pauls/java/heapdump.20061006.141801.9909.phd'
JVMDUMP010I Heap Dump written to /home/pauls/java/heapdump.20061006.141801.9909.phd
JVMDUMP007I JVM Requesting Java Dump using
'/home/pauls/java/javacore.20061006.141801.9909.txt'
JVMDUMP010I Java Dump written to /home/pauls/java/javacore.20061006.141801.9909.txt
JVMDUMP013I Processed Dump Event "uncaught", detail "java/lang/OutOfMemoryError".
Exception in thread "main" java.lang.OutOfMemoryError
        at java.util.HashMap.resize(HashMap.java:499)
        at java.util.HashMap.addEntry(HashMap.java:800)
        at java.util.HashMap.put(HashMap.java:441)
        at java.util.HashSet.add(HashSet.java:209)
        at MyLeakPgm.leak(MyLeakPgm.java:10)
        at MyLeakPgm.main(MyLeakPgm.java:18)
```

Figure 8-24 A heap dump is generated by IBM Technology for JVM automatically on OutOfMemoryError

Capturing a heap dump manually

It is often very useful to take heap dumps at specific times or intervals so that you can see changes in the Java heap over time. You can do this easily by sending a signal to the JVM, but it does require that you start the JVM with the **-Xdump:heap** command-line option.

Figure 8-25 shows how this can be done when invoking Java from within a Qshell session.

```
                          QSH Command Entry

  $
> java -Xdump:Heap SleepTimer 400
```

Figure 8-25 Setting the JVM to generate a heap dump

From a separate Qshell session you can use the ps utility to list all active processes running under a given user profile. Figure 8-26 shows an example. There is a job (204462/pauls/qp0zspwp in our example) that corresponds to the Java application. Note that the process ID (PID) is 298.

```
                          QSH Command Entry

$
ps -u pauls
     PID DEVICE       TIME FUNCTION       STATUS    JOBID
     299 -           00:00 pgm-ps         run       204463/pauls/qp0zspwp
     262 -           00:00 pgm-qzshsh     evtw      204350/pauls/qzshsh
     228 qpadev0006  00:00 cmd-qsh        deqw      204242/pauls/qpadev0006
     263 qpadev0008  00:00 cmd-qsh        dspw      204351/pauls/qpadev0008
     297 -           00:00 pgm-qzshsh     evtw      204461/pauls/qzshsh
     298 -           00:01 -              thdw      204462/pauls/qp0zspwp
$
kill -QUIT 298
$
```

Figure 8-26 Find the PID and start the Javadump

Figure 8-26 also shows how to use the **kill** command to generate the Javadump. A Javadump captures the current state of the JVM and can be automatically triggered under certain error conditions. The dump includes basic information about garbage collection, lock information, heap statistics, and a stack for each thread. In our example, (Figure 8-25) the JVM was started with the **-Xdump:heap** option, so that a Heap dump is generated in addition to the Javadump.

Important: Do not let the terms “kill” and “quit” confuse you. The VM continues running after the **kill** command is issued. However, generating a heap dump (when JVM is started with the **-Xdump:heap** command-line option) might take several minutes and it could have a noticeable impact on your application. For this reason it is advisable that you only generate heap dumps in a development environment. If the memory leak is only noticeable when running an application in a production environment, then you have to plan for a time that has the least impact to your business if an outage were to occur.

You might have noticed in Figure 8-26 that there is no confirmation message after the **kill** command is issued.

As you can see in Figure 8-27, the output of your JVM is updated to include the location and name of the dump files. This same information can also be found in the job log.

```
                              QSH Command Entry

java -Xdump:Heap SleepTimer 400
Going to sleep for 400 seconds...
JVMDUMP006I Processing Dump Event "user", detail "" - Please Wait.
JVMDUMP007I JVM Requesting Heap Dump using
'/home/pauls/heapdump.20061009.113847.298.phd'
JVMDUMP010I Heap Dump written to /home/pauls/heapdump.20061009.113847.298.phd
JVMDUMP007I JVM Requesting Java Dump using
'/home/pauls/javacore.20061009.113847.298.txt'
JVMDUMP010I Java Dump written to /home/pauls/javacore.20061009.113847.298.txt
JVMDUMP013I Processed Dump Event "user", detail "".
```

Figure 8-27 Showing the files generated when a heap dump is triggered

Deciding when to generate a heap dump

Because generating a heap dump can have a significant impact to the running application, you must try to generate as few as possible. Here are a couple of tips that might help:

- Increase the maximum heap size of the JVM to allow the leak to grow bigger.
- Enable verbose GC output (see 5.2.3, “Verbose GC output” on page 44).
- Analyze verbose GC output to observe how fast the free heap space is dropping.
- Do not take any dumps when the application is in its “startup” phase.
- Take four to six memory dumps during the period that the free heap space is dropping.

8.5.4 Using the Memory Dump Diagnostic for Java (MDD4J) tool

Now that you have generated some heap dumps you have to analyze them to see if you can get closer to finding a root cause of the memory leak. These dumps are much too big and cryptic to try any kind of manual analysis, therefore, a tool is necessary to process and summarize the data. There are a few tools that can do such analysis but we are going to focus on one called Memory Dump Diagnostic for Java (MDD4J). The MDD4J tool combines many of the best features from existing tools, such as Leakbot and HeapAnalyzer. IBM is currently committed to making MDD4J the primary tool for performing memory leak analysis.

Note: The Memory Dump Diagnostic for Java tool has the following recommendations for minimum hardware specifications:

- Approximately 5 GB of disk space
- 1.5 GB of physical memory (RAM)
- 2 GHz CPU (for Intel processors)

Installing MDD4J

MDD4J is available through the IBM Support Assistant tool. IBM Support Assistant (ISA) is the mechanism for delivering and maintaining tools such as MDD4J. If you do not have ISA installed or if you would like to read more about it, refer to Appendix B, “IBM Support Assistant” on page 181.

Launching MDD4J

With the tool successfully installed you are ready to analyze some heap dumps.

1. Launch IBM Support Assistant.
2. Click **Tools** on the IBM Support Assistant Welcome Page.
3. Click **WebSphere Application Server V6.1** under Products.
4. Click **Memory Dump Diagnostic for Java (MDD4J)** as shown in Figure 8-28.

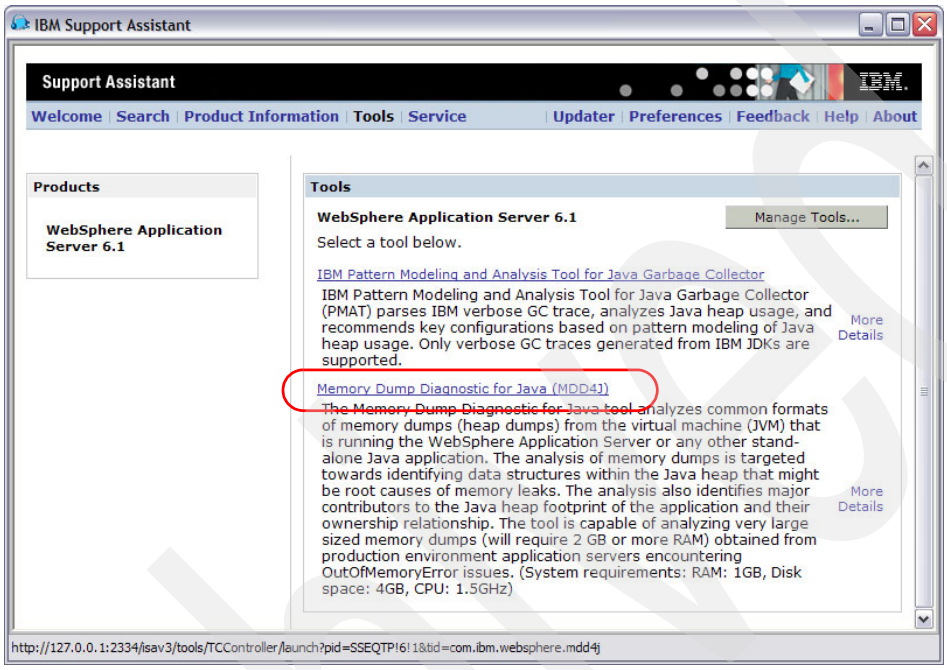


Figure 8-28 Launching MDD4J tool

5. MDD4J must launch and look like Figure 8-29.

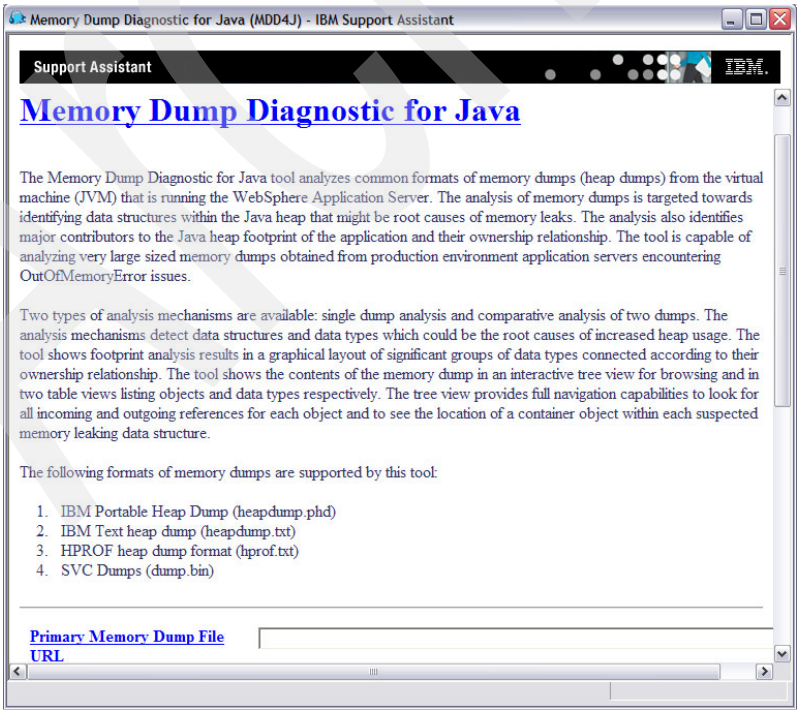


Figure 8-29 MDD4J start page

Analyzing data with MDD4J

MDD4J processes data and presents the results in the GUI. The following example helps illustrate how the analysis is done and how you can interpret those results. To get started Figure 8-30 shows a pictorial view of our Java program that leaks string objects.

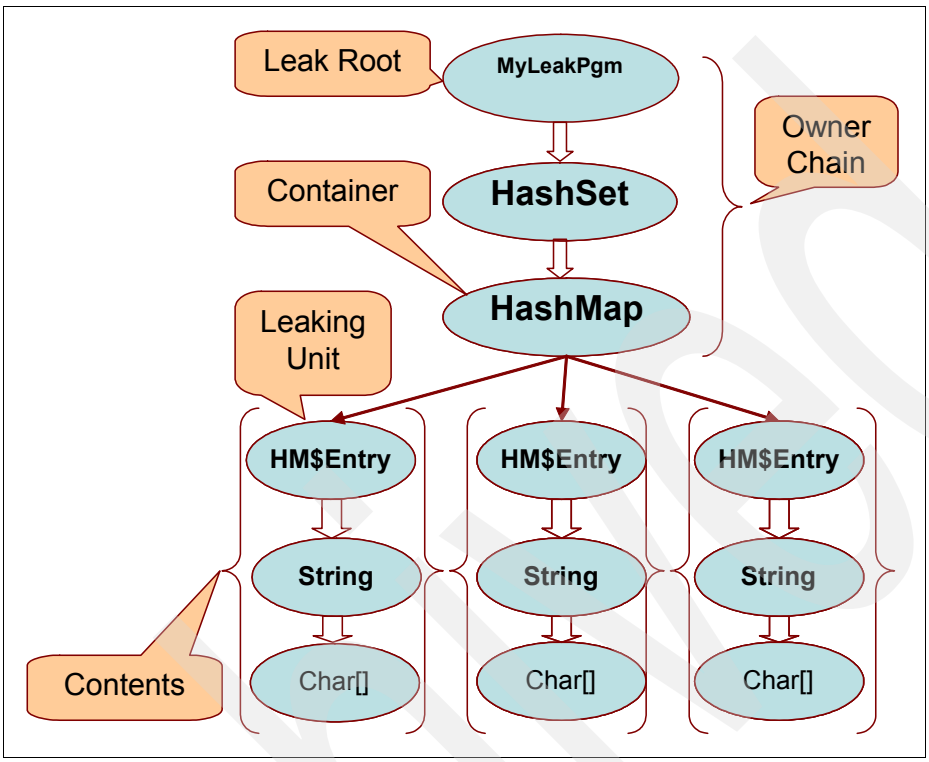


Figure 8-30 Showing the structure of the sample leak program

Defining the following terms help you understand Figure 8-30:

- *Leak root:* the class object which holds a static reference to the chain of objects leading to the leak container (MyLeakPgm)
- *Leak container:* the object (HashMap) holding all the leaking objects
- *Leak unit:* the data type that is the root of the data structures which are added to the enclosing Leak Container object (String Objects)
- *Owner chain:* is defined to be the chain of objects starting from a root

Notice in Figure 8-31, the MDD4J tool can accept two files as input. For a good comparative analysis, primary and baseline dumps are collected during a single run of a memory leaking application. The second file is optional and is used to do comparative analysis to the primary file. The primary dump refers to the dump taken after the memory leak has progressed considerably, consuming a large amount of the maximum configured heap size. The baseline dump is captured early on, when the heap has not yet been consumed significantly due to the memory leak. Because a comparative analysis typically yields the best results we show a comparative analysis in our example.

See Figure 8-31.

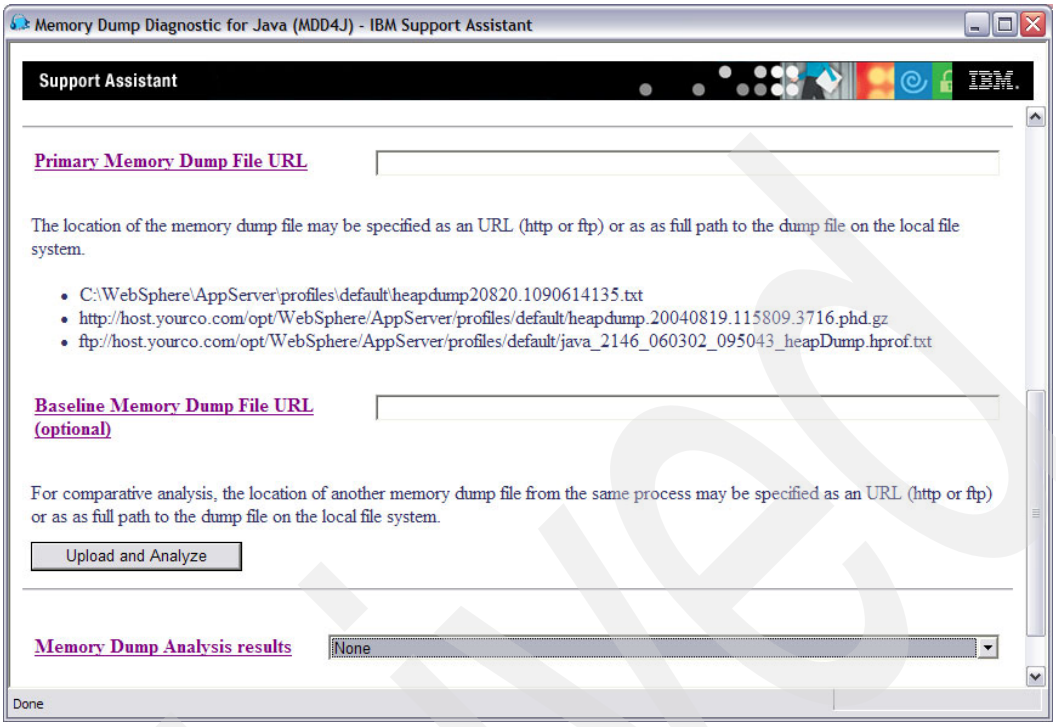


Figure 8-31 MDD4J can accept two files as input for comparative analysis

We generated two heap dumps using the instructions in “Capturing a heap dump manually” on page 112 while the sample leak program was running and specified them in the *Baseline* and *Primary* dump file fields. When you click the **Upload and Analyze** button you must see the screen shown in Figure 8-32.

Figure 8-32 The analysis status screen

Attention: The analysis could take a long time to finish. On a test PC (1.5 GHz, 1 GB of RAM) this example took about 32 minutes to complete. The primary dump file was about 42 MB and the baseline dump file was 15 MB in size. At the time of this writing the MDD4J development team was working on performance enhancements. All updates to MDD4J are delivered via IBM Support Assistant. It is recommended that you regularly check for updates and install any available updates.

If the analysis does not seem to be making any progress, you might want to look in the following directory to see if there are any errors in the log files:

```
<IBM Support Assistant install>/workspace/logs
```

The default workspace directory location is shown in Figure 8-33. Some potential errors are:

- The dump file was truncated or corrupted in some way
- There are too many objects in the dump or the size of the dump exceeded what MDD4J can process. Currently MDD4J can process dumps containing up to 30 million objects.

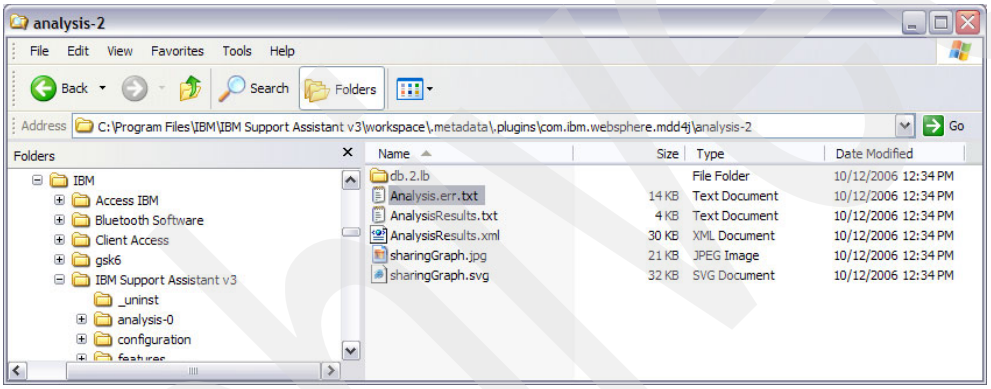


Figure 8-33 Location of the MDD4J log and results files

When the analysis does finish a summary screen is displayed as shown in Figure 8-34.

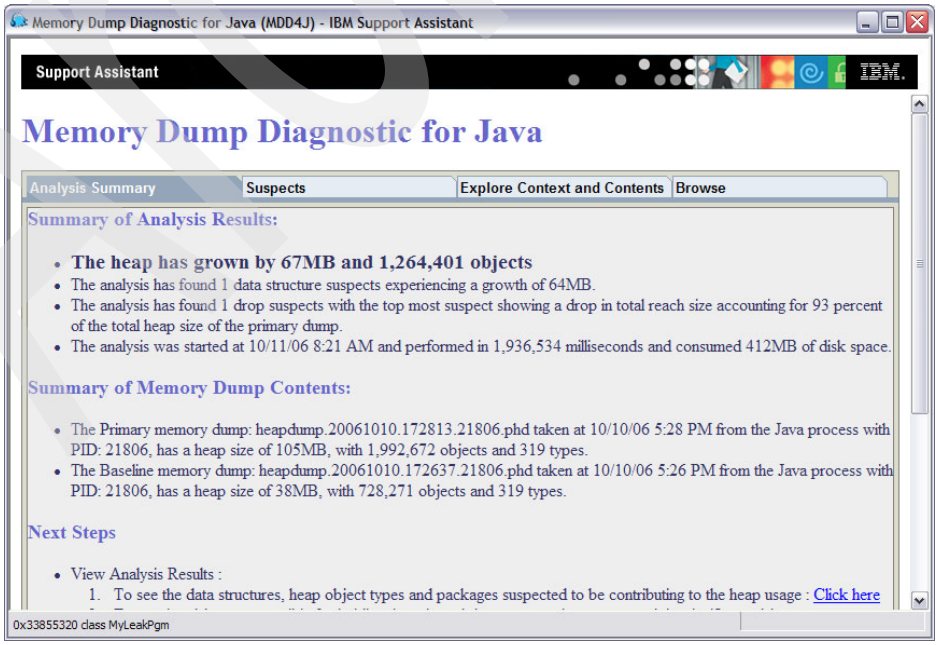


Figure 8-34 Summary of a comparative analysis

There are many different views of data and you probably find a few that you prefer as you work with the tool. Exploring the **Next Steps** section on the Analysis summary tab, shown in Figure 8-35, is a good way to see the various summaries and views available.

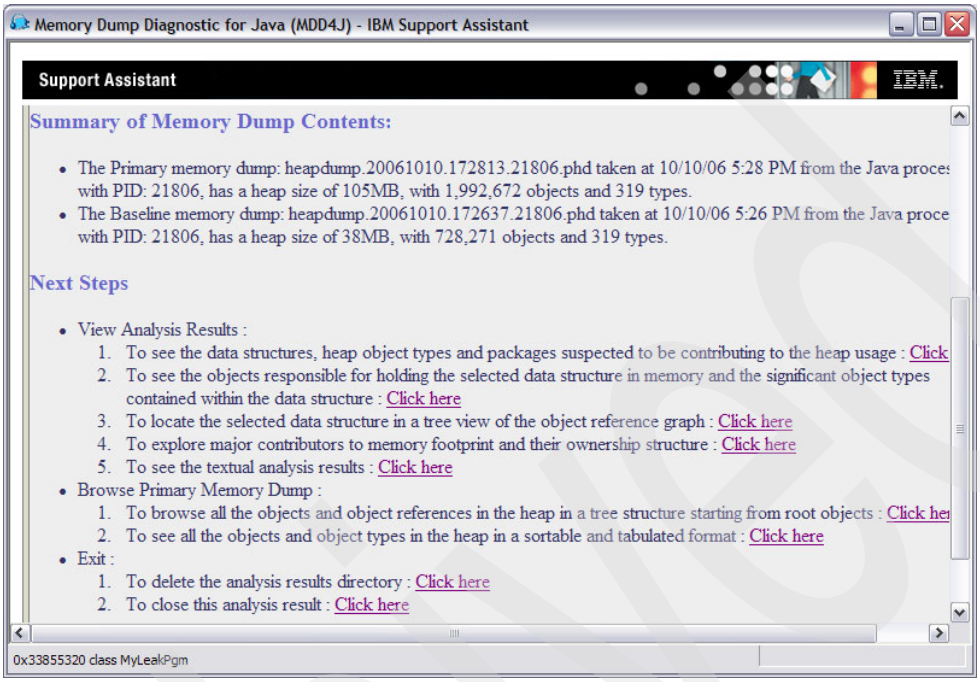


Figure 8-35 Following the Next Steps to view the analysis results

The Suspects view, shown in Figure 8-36, is a great place to start looking because, as the name implies, it indicates likely causes of a memory leak.

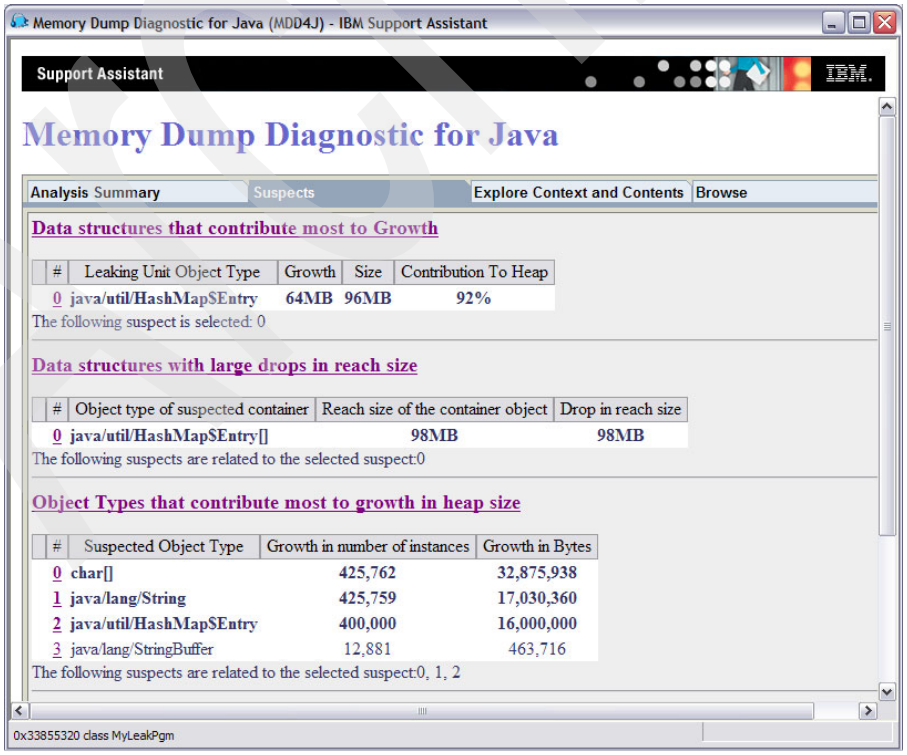


Figure 8-36 Showing suspected leak information

A graphical view is also available, as shown in Figure 8-37, by clicking the **Explore Context and Contents** tab.

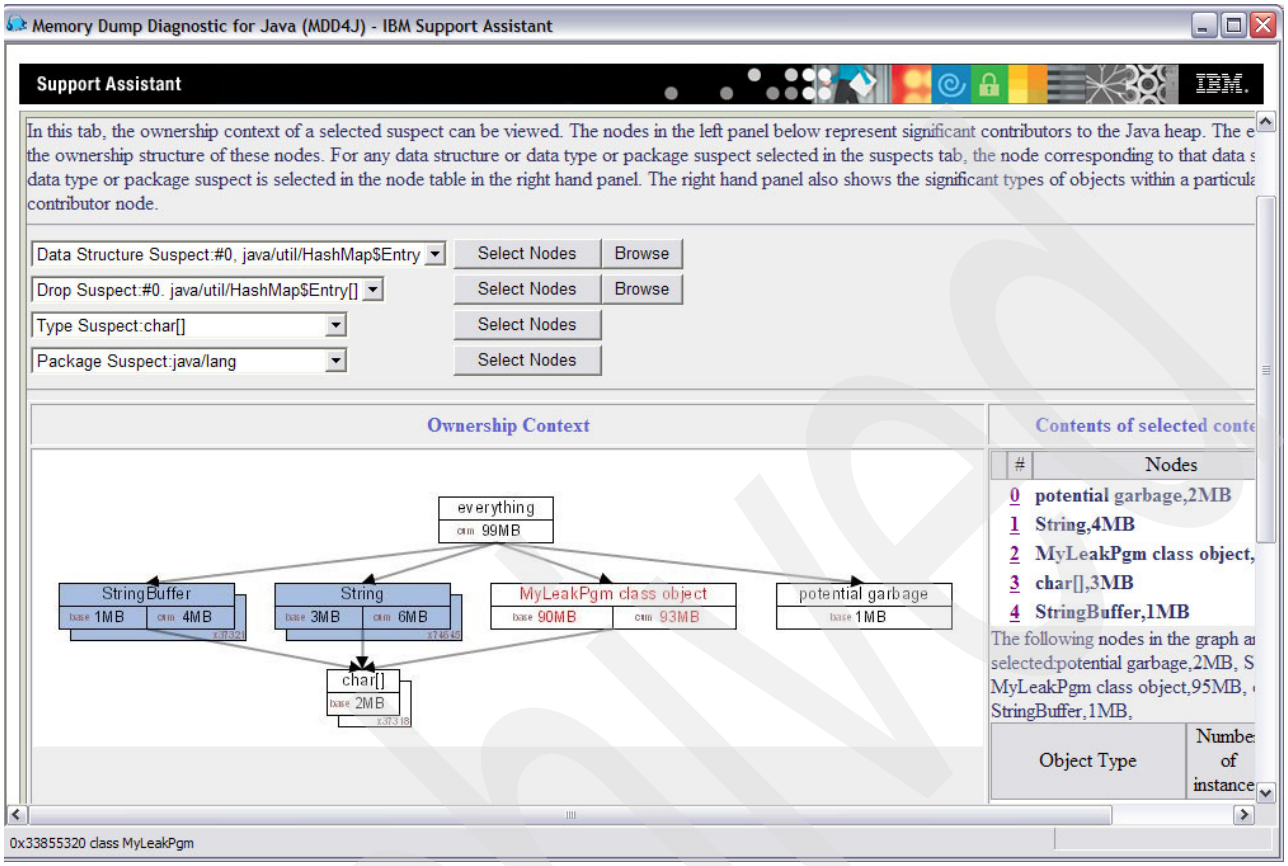


Figure 8-37 A graphical view of ownership context

You can also see details about selected objects in the tree by switching to the **Browse** tab.

In Figure 8-38 the details of the class MyLeakPgm are displayed. You can see that the total *reach size* is about 98 MB. This is a very big clue because you already know that the total heap size was about 105 MB when the second heap dump was generated, as shown in Figure 8-34 on page 118. This means that nearly all the heap space is “rooted” in the MyLeakPgm class.

This is a very obvious example of a memory leak but it is this type of information that would allow a Java developer to locate and fix the faulty code.

Note: Total reach size is defined as “The size of all objects which are reachable from the given object, in a single pass depth-first search of all the objects in the memory dump starting from all root objects where any object is visited at most once”.

See Figure 8-38.

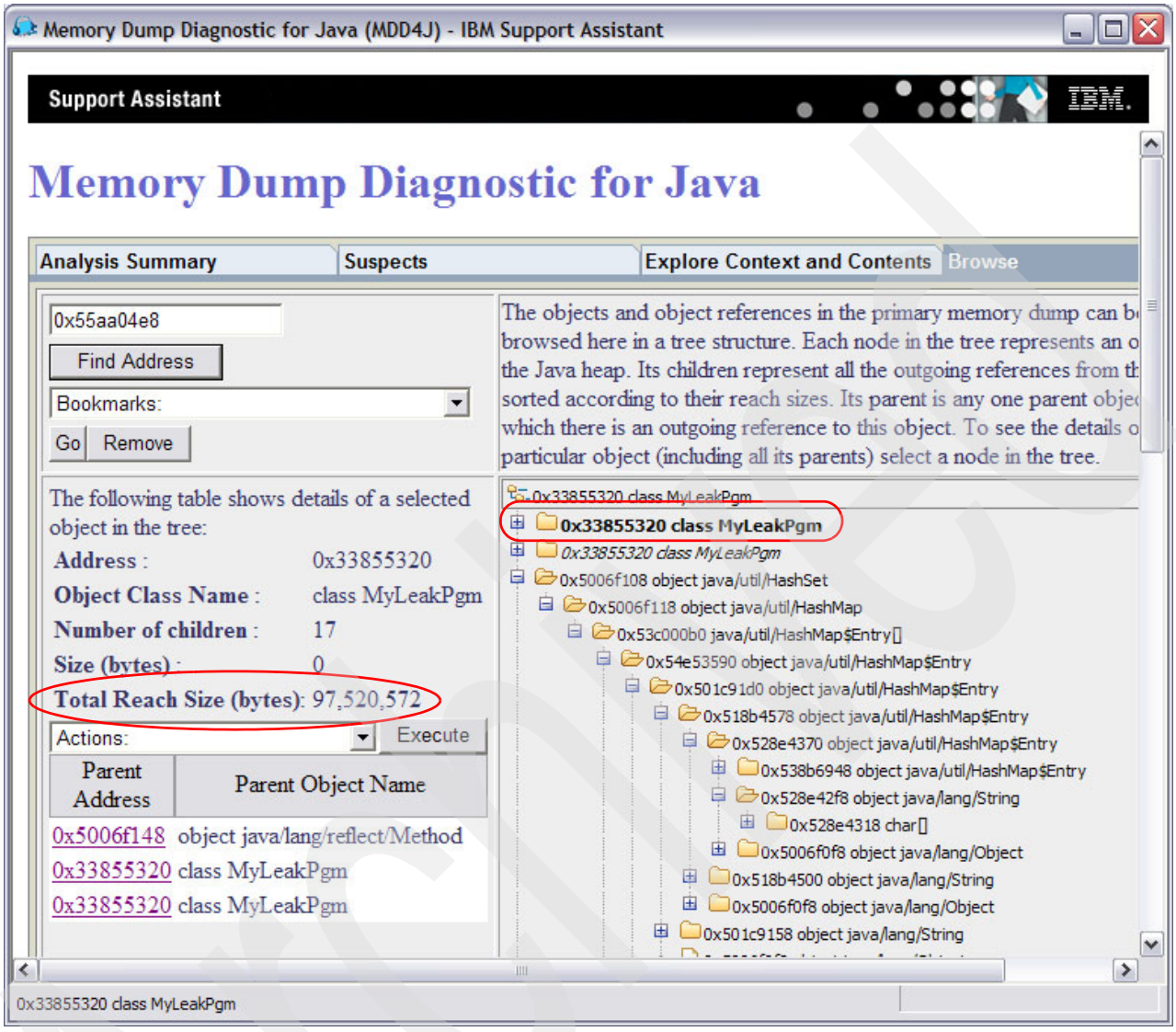


Figure 8-38 Showing details of selected objects in the tree

If you would like to view the heap contents directly you can bring up the object table in a sortable view. For example, you can view the object table by performing the following steps:

1. Start at the Analysis summary page, shown in Figure 8-34 on page 118, and click **To see all the objects and object types in the heap in a sortable and tabulated format: Click here**.
2. When the object table loads click the **Types Table** tab.
3. Click the **Growth in instances since Baseline dump** column.

   The object table is now sorted and showing the object types that have grown the most since the baseline dump was taken.

See Figure 8-39.

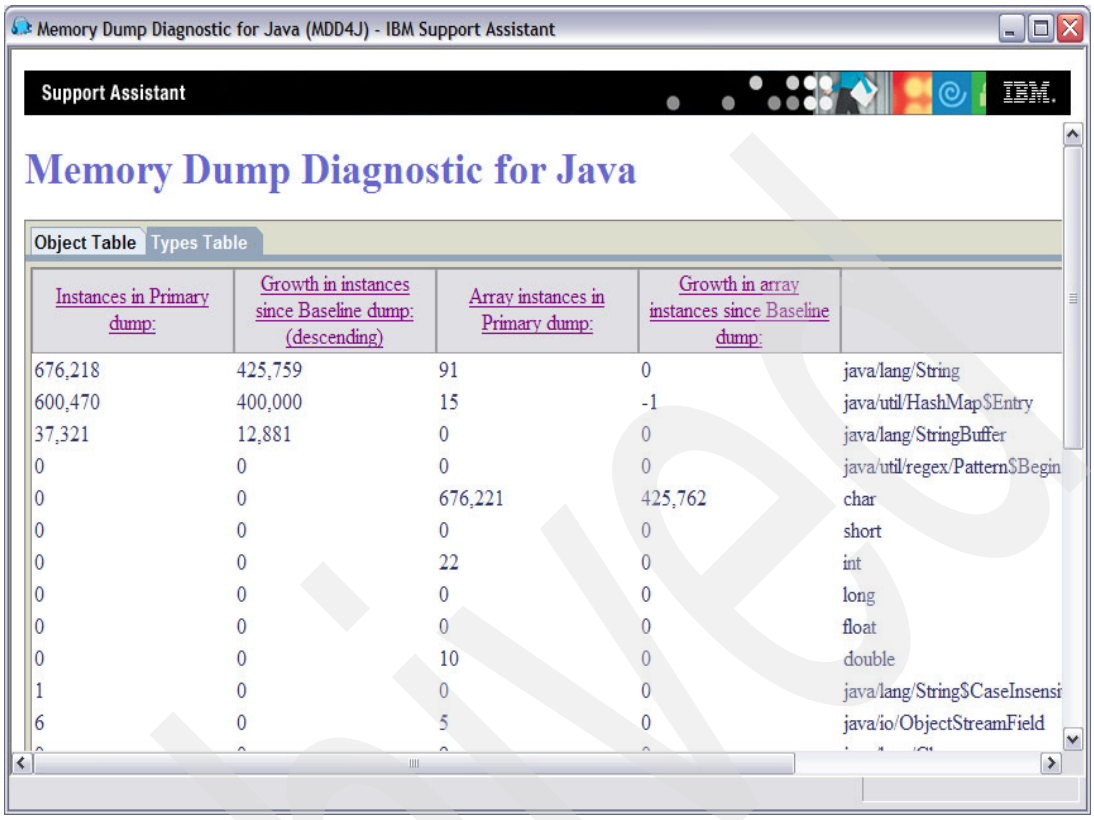


Figure 8-39 Showing the object table

The Future of MDD4J

As mentioned in 8.5.4, “Using the Memory Dump Diagnostic for Java (MDD4J) tool” on page 114, IBM is currently committed to supporting this tool as the primary customer tool for analyzing memory leaks. Enhancements and fixes for MDD4J are released on a regular basis, therefore, be sure that you are using the “Updater” feature of IBM Support Assistant to download and apply the fixes to your workstations. If you want to report a problem with MDD4J or any other IBM Support Assistant tool, use the IBM Support Assistant feedback option. For additional information about MDD4J visit the following Web site:

http://www-128.ibm.com/developerworks/websphere/library/techarticles/0606_poddar/0606_poddar.html

Analyzing memory leaks with WebSphere Development Studio Client for iSeries

WebSphere Development Studio Client for iSeries Advanced Edition has an option for analyzing memory leaks. It allows you to take snapshots of the memory heap automatically or manually as your application runs. You can then analyze the number of objects and memory usage growth during the interval. It includes several useful views, such as:

- Leak candidate view for identifying the objects most likely responsible for memory leaks
- Object reference graph, object references, and object details views provide information about objects consuming memory
- Statistical views display profiling statistics for packages, classes, and methods

In this section you can see how to identify a memory leak. We use the *Identify Java memory leaks sample* that comes with the tool.

Restriction: At the time of this writing, you can only use the Classic JVM for remote debugging in WebSphere Development Studio Client for iSeries version 6.0.1. However the approach presented in this section is applicable to IBM Technology for JVM when this is supported for remote debugging.

Perform the following steps:

1. Start WebSphere Development Studio Client for iSeries Advanced Edition on your workstation.
2. Open the Java perspective.
3. Locate the Java application project and source files for your application.
4. Right mouse click the class (which must have a main() method) and select **Profile** → **Java Application**.
5. At the properties window select the **Profiling** tab as shown in Figure 8-40).
6. Select the **Memory Leak Analysis - Manual heap dumps** option and click **OK**.

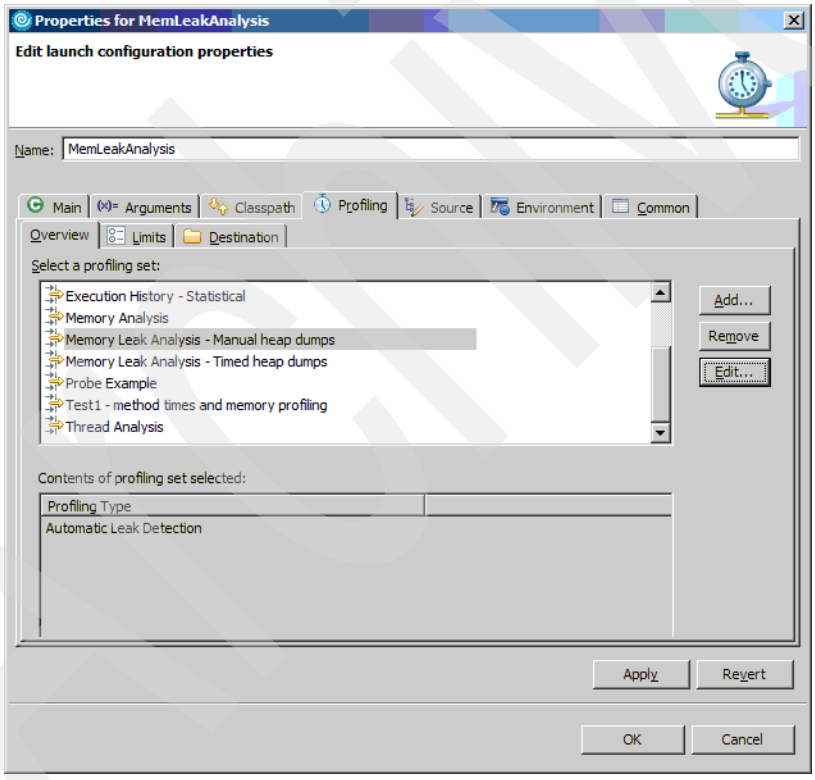


Figure 8-40 Memory leak analysis selection

7. If you do not have the profiling and logging perspective open, then you can see a confirm perspective switch? window open. If so, click **Yes**.
8. You must be in the profiling and logging perspective and see your application status as monitoring. If the console does not open go to the toolbar and select **Window** → **Show View** → **Console**. This displays any output data.

   You have to manually capture at least two heap dumps to do memory leak analysis.

9. After the application has “warmed up”, you have to capture a heap dump. Click the **Capture Heap Dump** button as shown in Figure 8-41.

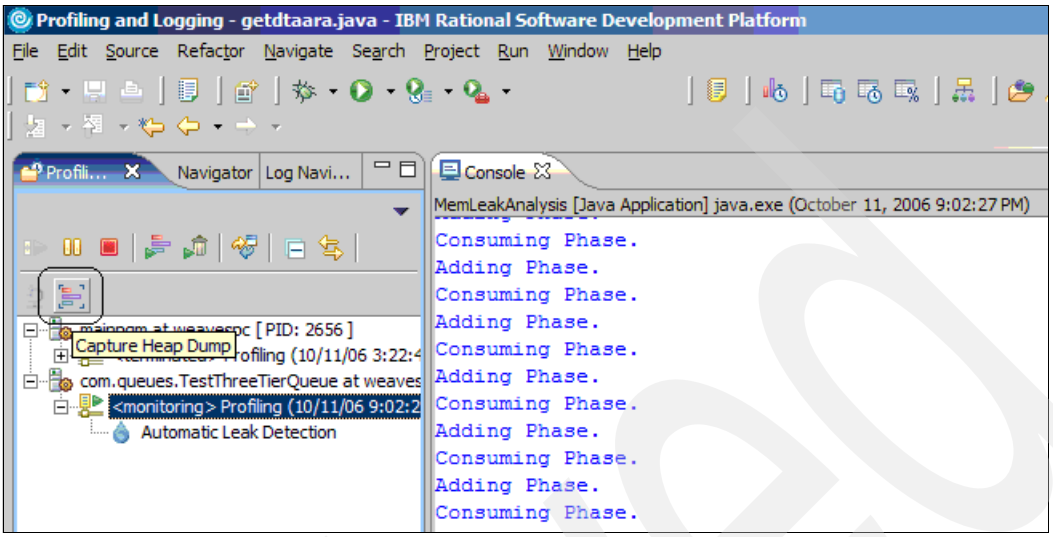


Figure 8-41 Manually capture a JVM heap dump

10. You must see a heap dump listed under your running application. Let the application continue running.
11. After the application has run a few more times, click the **Capture Heap Dump** button again.
12. You must see two heap dumps, as shown in Figure 8-42. Click the **Terminate** button to stop the application and the profiling.

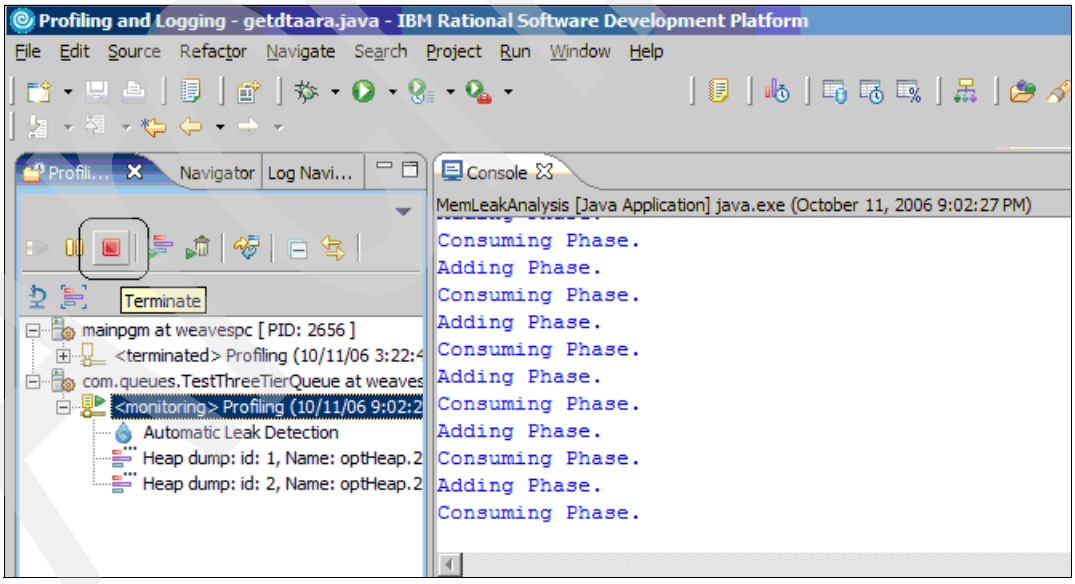


Figure 8-42 heap dumps available for analysis

13. You must see the leak candidates view. Click the **Analyze for Leaks** button. See Figure 8-43.

> **Note:** The memory leak analysis tools use technology from MDD4J. You can import and analyze heap dumps from standard heap dumps (using the IBM_HEAPDUMP environment variable, an HPROF heap dump, an OS/390® heap dump, or a Hyades optimized heap dump.

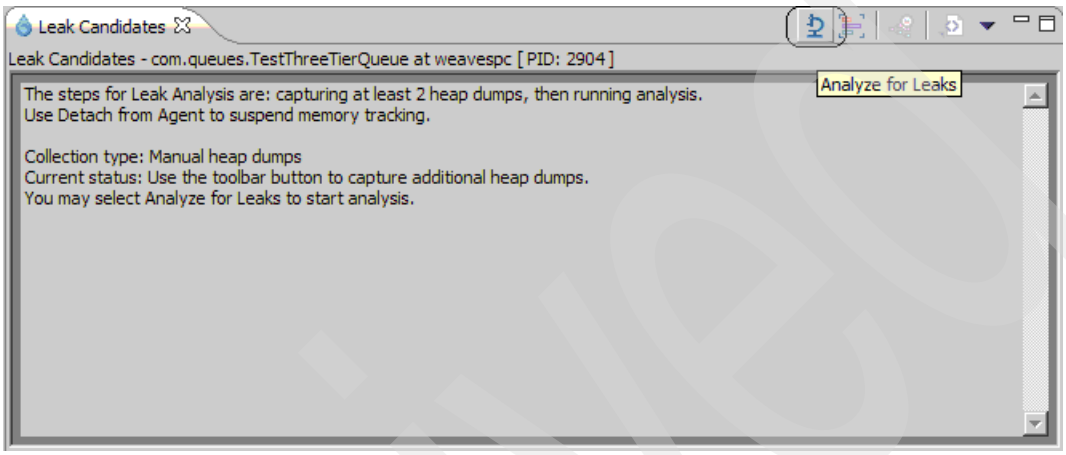


Figure 8-43 Analyze the heap dumps for memory leaks

14. At the “select leak analysis options” window you must see your two heap dumps selected. You must also see the threshold set to 20 (the default). Setting the number low attempts to find many leaks. Setting the number high attempts to find fewer. In this example the default is used. Click **OK** after specifying the appropriate heap dumps and threshold.

    It might take a few minutes to analyze the heap dumps. You must see the progress statistics in the lower right part of the window, as shown in Figure 8-44.

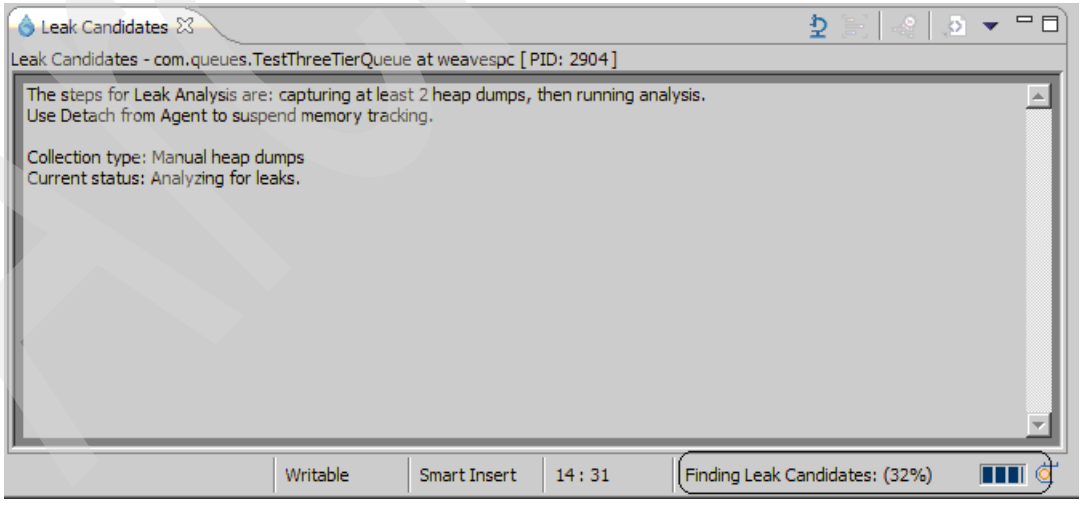


Figure 8-44 Analyzing memory leak candidates

If one or more memory leaks have been detected, they are listed in the leak candidates view. The higher the likelihood value, the higher the probability that a memory leak has been detected. You must also see the parent application, the object type causing the leak and the amount of memory.

In this example you see from Figure 8-45 that a vector is holding on to references for String objects.

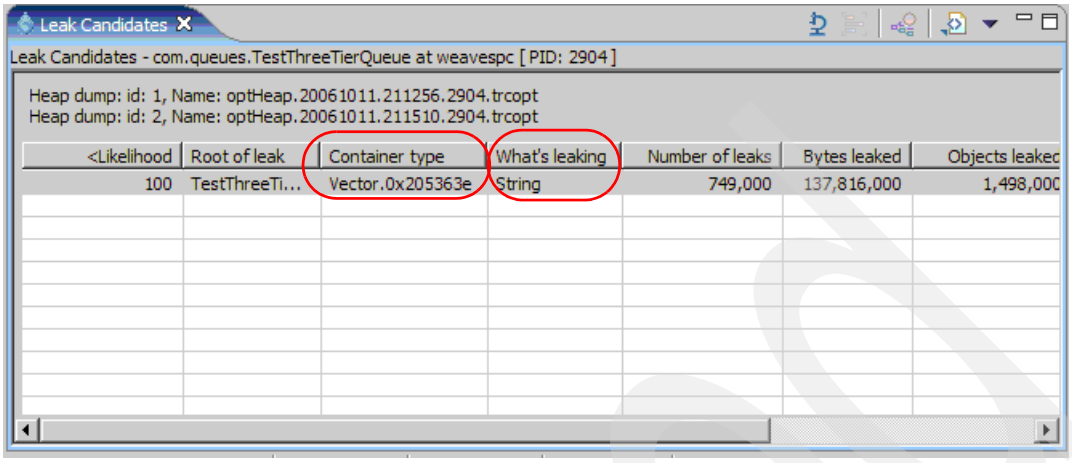


Figure 8-45 Memory leak candidate identified

15. Double click the line with the leak candidate data entries. You must see an object reference view open, with the “path” to the offending container and object. If you hover the mouse over an object or the link, you see statistics on the number of objects and memory leaked, as shown in Figure 8-46.

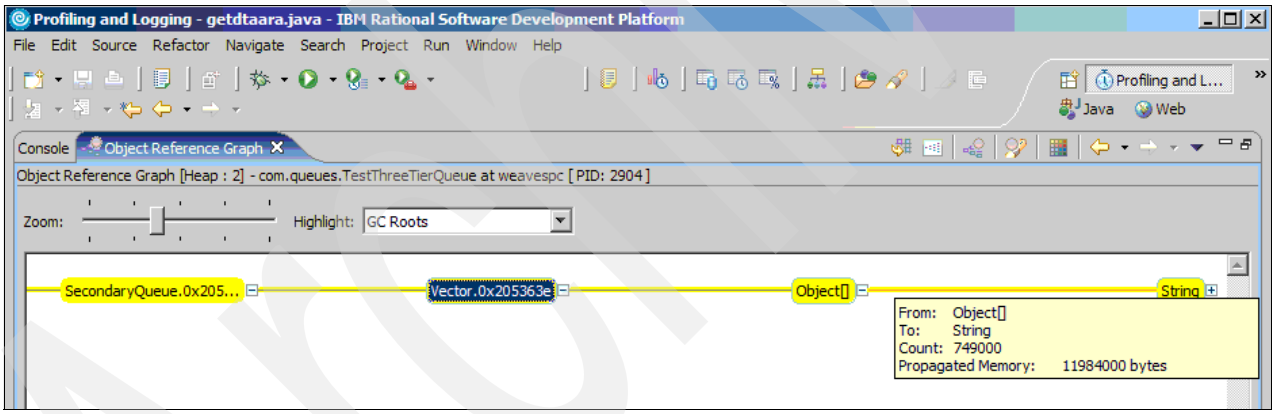


Figure 8-46 Object reference view of the potential memory leak

In this example the memory leak is in the SecondaryQueue class, from a vector object that is retaining references to String objects.

16. The next step is to go back to the Java perspective and open up the source file for the offending class, as shown in Example 8-2. This is the offending code. myQ is a vector object and retains references to all objects that it contains.

Example 8-2 Source code

```
private Vector myQ;
private int currentPos;
public SecondaryQueue(){
   myQ = new Vector();
   currentPos=0;
}
```

```
public void add(Object obj){
   myQ.add(obj);


public Object getNext(){
   // First In First Out.
   // Get an object exactly once,
   // currentPos keeps track of last item removed.
   if(myQ.size()>currentPos){
      currentPos++;
      return myQ.get(currentPos-1);
   }
   return null;
}
```

17. Look at the `add(Object obj)` method. Each invocation of this method adds an object to the myQ vector. These could be String or other objects. As objects are added to the vector, the memory usage increases accordingly.
18. Look at the `getNext()` method. This returns the object at `currentPos`, but the object still stays in the myQ vector. This is a form of a memory leak. In order to properly remove an object from Vector, one must use the remove method:

    `public E remove(int index)`

    This method removes an object from the Vector and returns it to the caller.

    The next code snippet shown in Example 8-3 illustrates how to use this to fix the memory leak. In this case the object is removed from the Vector after the getNext() method. Now the myQ vector object cannot grow indefinitely like it could in the previous example.

Example 8-3 Correct implementation of the getNext method

```
public Object getNext(){
     if(myQ.size()>0){
     return myQ.remove(0);
     }

     return null;
}
```

8.6 Performance issues

The focus of this book is JVM specific issues. Other performance issues related to an application or middleware product such as WebSphere Application Server are not discussed here, but we provide links to more detailed resources related to these performance issues.

8.6.1 Quick fix: general guidelines

Creating a JVM process to run Java applications on System i platform requires memory and process resources. For short-running applications it takes longer to create the JVM process than the actual application run time. Middleware products such as WebSphere Application Server create a JVM process and reuse it for all runtime operations.

A common approach to minimizing the impact of JVM creation for System i platform is to write the Java application such that it listens for requests on an i5/OS data queue and places any output on another data queue. In other words, the requesting application sends a message to a data queue with the appropriate input parameters, rather than calling the application directly.

Generally speaking, performance problems can be caused by the client, network, or server. Common client problems include old hardware, inadequate memory, missing fixes, and poorly written code. Common network problems include inadequate bandwidth, latency, excessive broadcasting or file transfer activity, and poor design. Your first step in dealing with perceived performance issues must be to investigate whether the client or network is an issue, and then look at potential issues at the server level.

8.6.2 Server guidelines

These are the major categories of server based contributors to overall Java application performance on System i platform:

- Hardware
- Operating system and system values
- Database or other i5/OS resources
- Coding techniques
- Java virtual machine

The Java virtual machine guidelines are covered next in 8.6.3, “Java virtual machine” on page 132.

Hardware

General guidelines for hardware include having adequate processor, memory, disk, and communications resources. System i IBM POWER™ 5 and IBM POWER 5+ based servers provide faster response time and more throughput than previous generation IBM POWER 4 based servers. You can use i5/OS commands such as WRKACTJOB, WRKSYSSTS, and WRKDSKSTS to perform real time sampling to understand processor, memory, and disk usage on your System i platform. You can also use i5/OS Performance Collection Services to perform long term monitoring.

You can use the IBM Systems Workload Estimator (WLE) to size a new system, a proposed upgrade to an existing system, or a consolidation of several systems. The WLE tool currently does not have the capability to size generic Java workloads. However, it can size WebSphere Application Server-based workloads and provide basic guidelines on processor, memory, and disk requirements for a given workload. WLE is available at no charge on the Web at:

`http://www-912.ibm.com/supporthome.nsf/document/16533356`

Java applications tend to consume more memory and processor resources than compiled applications written in RPG or COBOL. Java applications also tend to use more high level language instructions than traditional System i applications and benefit from newer System i models. The IBM Accelerator for System i5 is available on selected System i5 520 models and can also significantly improve Java application performance. The accelerator is a software feature code that enables the entire processor’s capacity to be available for batch processing workloads. For example, feature number 7354, available for the System i 7143 520 Express Configuration, increases the commercial processing workload (CPW) rating for System i platform from 1200 to 3800. Another key recommendation is for Java based applications to be run on systems with L3 cache. Refer to the *IIBM System i5 Handbook IBM i5/OS Version 5 Release 4 January 2006*, SG24-7486.

For additional recommendations visit the System i performance management Web site. It includes links to generic System i performance resources such as benchmark results and specific articles on Java and WebSphere Application Server for i5/OS. It also has links to the i5/OS performance capabilities and references guides, which provides extensive tips and techniques for optimizing overall System i performance. These resources are available at:

http://www-03.ibm.com/servers/eserver/iseries/perfmgmt/resource.html

Operating system and system values

As mentioned in the previous section, Java applications tend to consume more processor and memory resources than traditional System i applications. Some of this is an inevitable trade-off because of the code portability Java applications provide. More importantly, Java applications can provide function and integration that is difficult or impossible with traditional System i applications. This must more than compensate for the additional system resource requirements.

Running Java based applications on System i platform might require you to change your i5/OS work management and tuning strategies. Java is inherently a multithreaded environment and the JVM spawns threads based on how the application was written and other runtime events. Java based middleware such as WebSphere Application Server also makes extensive use of threads for its internal operations, and also accessing System i applications and data resources. An execution thread on System i platform is represented as an activity level. Activity levels can be specified at multiple levels (memory pool, subsystem, and so on).

In general, the activity level must be high enough to avoid wait → ineligible and active → ineligible transitions, but not so high as to cause system instability. You can use the i5/OS WRKSYSSTS command to view the current activity level in your memory pools. Refer to the Max Active column shown in Figure 8-47.

```
                          Work with System Status                   RCHAS60
                                                          10/09/06  11:23:38
% CPU used . . . . . :          .6    Auxiliary storage:
% DB capability  . . . . :      .0      System ASP . . . . . :     492.3 G
Elapsed time . . . . . :   00:00:00     % system ASP used  . . :    23.4057
Jobs in system . . . . . :       979     Total  . . . . . :        597.8 G
% perm addresses . . . . :      .010     Current unprotect used :    8559 M
% temp addresses . . . . :      .028     Maximum unprotect  . . :    9162 M

Type changes (if allowed), press Enter.

System    Pool    Reserved    Max   Active->  Wait->  Active->
 Pool   Size (M)  Size (M)  Active    Wait     Inel     Inel
   1     1394.55    691.16   +++++      .0       .0       .0
   2    25107.60      4.20     590    1132       .0       .0
   3     3193.59       .00     299   125.7       .0       .0
   4         .25       .00       5      .0       .0       .0

                                                                    Bottom
```

Figure 8-47 WRKSYSSTS screen shows the size and activity levels for each memory pool

By default, Java applications run in the System i *BASE memory pool. Therefore, you must ensure that the activity level is set appropriately, and that there is adequate memory in the pool. This can be influenced (positively or negatively) by the setting of the Performance adjustment (QPFRADJ) system value. If this is set to a value other than 0, i5/OS attempts to periodically move memory and activity levels between memory pools. This might or might not be advantageous for your particular environment. A common recommendation is to appropriately tune the memory pool size and activity level for all workloads on your systems, then set the QPFRADJ system value to 0 (no adjustment). You can then monitor your system resource usage, throughput and other performance results and make changes as appropriate. However, this might not always be possible in production environments.

A common technique is to isolate Java (especially WebSphere Application Server) based workloads into their own memory pool. This can be particularly advantageous in environments where multiple workloads are competing for the same processor, memory, and activity level resources. i5/OS allows you to create a shared or private memory pool and define the pool size and activity level to ensure your Java applications always have an appropriate level of resources. i5/OS also allows you to create your own subsystems and define routing entries that specify the memory pools to be used for specific tasks. This is also popular on System i platform, especially with WebSphere Application Server based workloads. It is likely that you have to evaluate the feasibility of doing this in your environment on a case by case basis, especially if you have limited memory resources on your System i platform.

There are many other i5/OS tuning tips and techniques you can use, such as ensuring the prestart jobs for database requests (job QSQSRVR in the QSYSWRK subsystem and job QZDASOINIT in the QUSRWRK subsystem) are set high enough. For a comprehensive list of recommendations on i5/OS tuning for Java workloads, refer to chapter 5 of the IBM Redbook titled *Maximum Performance with WebSphere Application Server V5.1 on iSeries*, SG24-6383.

Database and other i5/OS resources

Another key element in obtaining the best possible performance from Java applications running on i5/OS, is to ensure you have optimized access to resources such as DB2 for i5/OS. This includes optimizing:

- Connection to the database
- Java database connection (JDBC) and SQL code
- Database design

Generally speaking, Java based applications use a Java Database Connectivity (JDBC) driver to connect to relational database resources such as DB2 for i5/OS. There are two JDBC drivers available for DB2 for i5/OS:

- The native driver
- The IBM Toolbox for Java driver

Generally speaking, the toolbox driver is more flexible but the native driver performs better. Also, an XA enabled JDBC driver (for transactional integrity such as two phase commit) imposes additional overhead compared with a non-XA enabled JDBC driver.

Table 8-1 provides a comparative analysis of two JDBC drivers.

Table 8-1 Comparison between the System i native and toolbox drivers

Attribute	Native JDBC driver	IBM Toolbox for Java JDBC driver
Implementation	Database and Java application must be on the same machine or partition.	Database and Java application can be on the same or different partitions.
Packaging	Packaged with product 5722-JV1.	Packaged with product 5722-JC1 or available at: http://jt400.sourceforge.net/
Application usage	Can be used with JDBC driver manager or data source connections.	Can be used with JDBC driver manager or data source connections.
Transaction integrity	XA and non-XA enabled versions available.	XA and non-XA enabled versions available.
i5/OS job structure	Uses QSQSRVR jobs in QSYSWRK subsystem.	Uses QZDASONIT jobs in QUSRWRK subsystem.
Interface to DB2 for i5/OS	Uses direct call level interface to DB2 for i5/OS.	Uses database host server job and TCP/IP sockets interface to DB2 for i5/OS.

Coding techniques

Another key to optimizing performance is coding techniques related to DB2 access. For overall Java coding recommendations refer to chapter 9 in the IBM Redbook titled *Maximum Performance with WebSphere Application Server V5.1 on iSeries*, SG24-6383.

Using a data source connection within your application (rather than the JDBC driver manager) allows you to take advantage of database connection pooling. Connection pooling provides a set of reusable i5/OS database access jobs that improves scalability. Connection pools also facilitates the reuse of prepared statements and open data paths, which can also improve scalability. Both the native and toolbox JDBC drivers support connection pooling.

Stored procedures are another option for improving database access performance. Stored procedures represent callable programs on the System i platform, which improve performance in several ways. Stored procedures help reduce the number of calls between the application and the DB2 for i5/OS runtime environment. Stored procedures also minimize the overhead associated with dynamic Structured Query Language (SQL) such as parsing and validating SQL statements, and creating an access plan. The toolbox JDBC driver supports SQL packages, which can also improve overall scalability of your applications. Efficient SQL coding techniques, such as only accessing the data you require, are also essential. For examples, refer to section 9.2.4 of the IBM Redbook titled *Maximum Performance with WebSphere Application Server V5.1 on iSeries*, SG24-6383.

Finally, ensure you have good database design concepts in place, such as indexes. DB2 for i5/OS supports binary radix indexes, and also encoded vector instances. Depending on your specific database environment and SQL statements used in your applications, one might be much preferable over the other. You can use the Visual Explain tools (part of the iSeries Navigator product) to create and analyze database monitor jobs on your System i platform.

iSeries Navigator tools also include index advisors, which suggest the most appropriate types of indexes over your table columns. You can also use the Visual Explain tools to model and optimize your SQL statements. Additionally, ensure you are using the latest versions of the DB2 for i5/OS group PTF package (SF99504 on V5R4M0).

For more detailed information about DB2 for i5/OS performance, the following publications are helpful:

- *Preparing for and Tuning the SQL Query Engine on DB2 for i5/OS*, SG24-6598
- *Stored Procedures, Triggers, and User-Defined Functions on DB2 Universal Database for iSeries*, SG24-6503
- *SQL Performance Diagnosis on IBM DB2 Universal Database for iSeries*, SG24-6654

8.6.3 Java virtual machine

This section considers how JVM features such as JIT compilation and garbage collection can influence application performance. Garbage collection is an area which can influence application performance to some extent. Chapter 5, “Tuning the garbage collector” on page 35 discussed how to adjust garbage collection settings appropriate to your environment. This includes steps such as setting the initial heap size and maximum heap size to appropriate values. This section takes an example of an application that is performing poorly. We show how to investigate if inappropriate garbage collection settings are having a negative impact on application performance and how to confirm that the problem is resolved.

It is worth repeating that for the majority of applications the default GC policy gives satisfactory performance. Adjusting the GC policy must be done only when your application performance does not meet your goals. Keep in mind though that you have to perform tuning activities at the application level first, for example, based on application profiling.

The performance gain to be achieved by using an alternative garbage collection policy varies by application, but can be quite small. In some cases a particular GC policy might in fact have a negative impact on application performance.

General guidelines

The IBM Technology for JVM has many built-in performance features, such as an advanced JIT compiler that optimizes code. For example, you do not have to “tune” the JIT threshold parameter. In Classic JVM, you can set the os400.jit.mmi.threshold=xyz custom property to a value other than the default of 2000, however, this applies to the entire JVM and is not widely used in practice. Refer to chapter 5 of IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide, for details on the IBM Technology for JVM JIT optimization.

An important aspect of JVM performance is to ensure your applications do not have a memory leak or that they do not indiscriminately create objects. A memory leak occurs when objects improperly hold on to references, such as can happen in recursive calls. It can cause memory usage to continue growing over time even though the workload has reached a steady state condition. Refer to 8.5, “Investigating a suspected memory leak” on page 111 for more details.

Verbose garbage collection typically imposes modest overhead and provides useful JVM runtime information. However, verbose class loading and verbose JNI impose overhead and are most useful for debugging, rather than runtime performance analysis. You can also adjust the size of your shared class cache, as described in 6.2.2, “Deploying shared classes” on page 54.

Some other general recommendations are:

- Do ensure you are using the latest Java group PTF package (SF99291 for i5/OS V5R4M0)
- Do not disable the JIT compiler unless performing troubleshooting
- Do not enable JVM profiling or other tracing unless it is absolutely necessary
- Do not disable class garbage collection (JVM argument *-Xnoclassgc*)

> **Important:** Be very careful when you specify JVM command line arguments. Mistakes as simple as case sensitivity in the parameters can prevent the JVM from starting.

Troubleshooting garbage collection

5.2.3, “Verbose GC output” on page 44 explained how to manually analyze verbose GC output to support heap analysis and tuning activities. The IBM Support Assistant provides a verbose GC data visualizer tool which you can use for such a task; the Extensible Verbose GC Toolkit (EVTK).

Overview of EVTK

EVTK parses and plots verbose GC logs and -Xtgc output. It provides graphical display of a wide range of verbose GC data values and at the time of writing is compatible with optthruput, optavgpause, and gencon GC policies.

You can use EVTK to support GC tuning and also general GC analysis and troubleshooting. We present a scenario where EVTK is used to analyze verbose GC output from a WebSphere Application Server environment which is not performing as well as expected. After identifying the possible cause of the performance bottleneck from the EVTK analysis, we make changes and use EVTK to confirm that the problem has been resolved.

> **Note:** At the time of writing this book IBM was planning to make EVTK available in IBM Support Assistant.

The scenario presented here involved a WebSphere Application Server V6.1.0.1 Base server with the Trade 6 performance sample application installed.

Trade 6 application

IBM Trade Performance Benchmark Sample for WebSphere Application Server (otherwise known as Trade 6) is the WebSphere end-to-end benchmark and performance sample application. The new Trade benchmark has been re-designed and developed to cover WebSphere's significantly expanding programming model. This provides a real world workload driving WebSphere's implementation of Java 2 Enterprise Edition (J2EE) 1.4 and Web Services, including key WebSphere performance components and features.

Test scenario

To stress the application server we generated a simulated workload for the Trade 6 application using Rational® Performance Tester V6.0.0.1. We configured verbose GC output prior to starting WebSphere Application Server by specifying the following filename in the Generic JVM arguments field:

```
verbosegclog:filename
```

Refer to “Configuring the WebSphere Application Server JVM” on page 173 for instructions on setting JVM arguments.

We used the default optthruput GC policy and initially ran the scenario with artificially small heap size settings so that the garbage collector would have to work very hard to prevent the heap from running out of space completely:

```
-Xms35m -Xmx60m
```

These parameters were also set in the *Java Virtual Machine* settings of our test server.

The resource constraints in this scenario are clearly highly exaggerated for the purposes of demonstrating the usage of EVTK for analyzing verbose GC data. Correspondingly, the results are not typical, but show that for a highly-loaded and constrained application, GC pauses can have a measurable impact on application throughput and response time, as you might expect. EVTK allows us to determine that on this occasion the cause of poor performance is related to increased garbage collection due to insufficient resources.

The server was placed under load by Rational Performance Tester for ten minutes and the verbose GC log collected for analysis. Because garbage collection cycles occurred prior to test start, they were not due to the test workload, data for these collections was manually removed from the verbose GC output before analysis with EVTK. This was necessary so that the graphs produced by EVTK and Rational Performance Tester covered the same period and could therefore be compared like-for-like.

It is possible to change the period of verbose GC data displayed in EVTK, but you would have to adjust both the start and end times displayed through EVTK to correlate to the start of the test (as opposed to the startup of the JVM) for the same reason. We felt it was more accurate to edit the verbose GC log file manually because we were able to refer to the time stamp in each GC cycle entry and tell for certain whether it occurred during the test period.

Viewing verbose GC data in EVTK

At the time of this writing, EVTK was not yet integrated into IBM Support Assistant and was supplied as a JAR file.

To view the verbose GC data in EVTK, we used the following procedure:

1. Copy the verbose GC output log file from i5/OS to a PC with the IBM Support Assistant and EVTK installed.
2. Start EVTK by running the vtk.bat file.
3. Select **File** → **Open File**.
4. Locate the verbose GC log file to be analyzed then click **Open**.

   EVTK processes the verbose GC log and displays a default chart from the data, as shown in Figure 8-48.

See Figure 8-48.

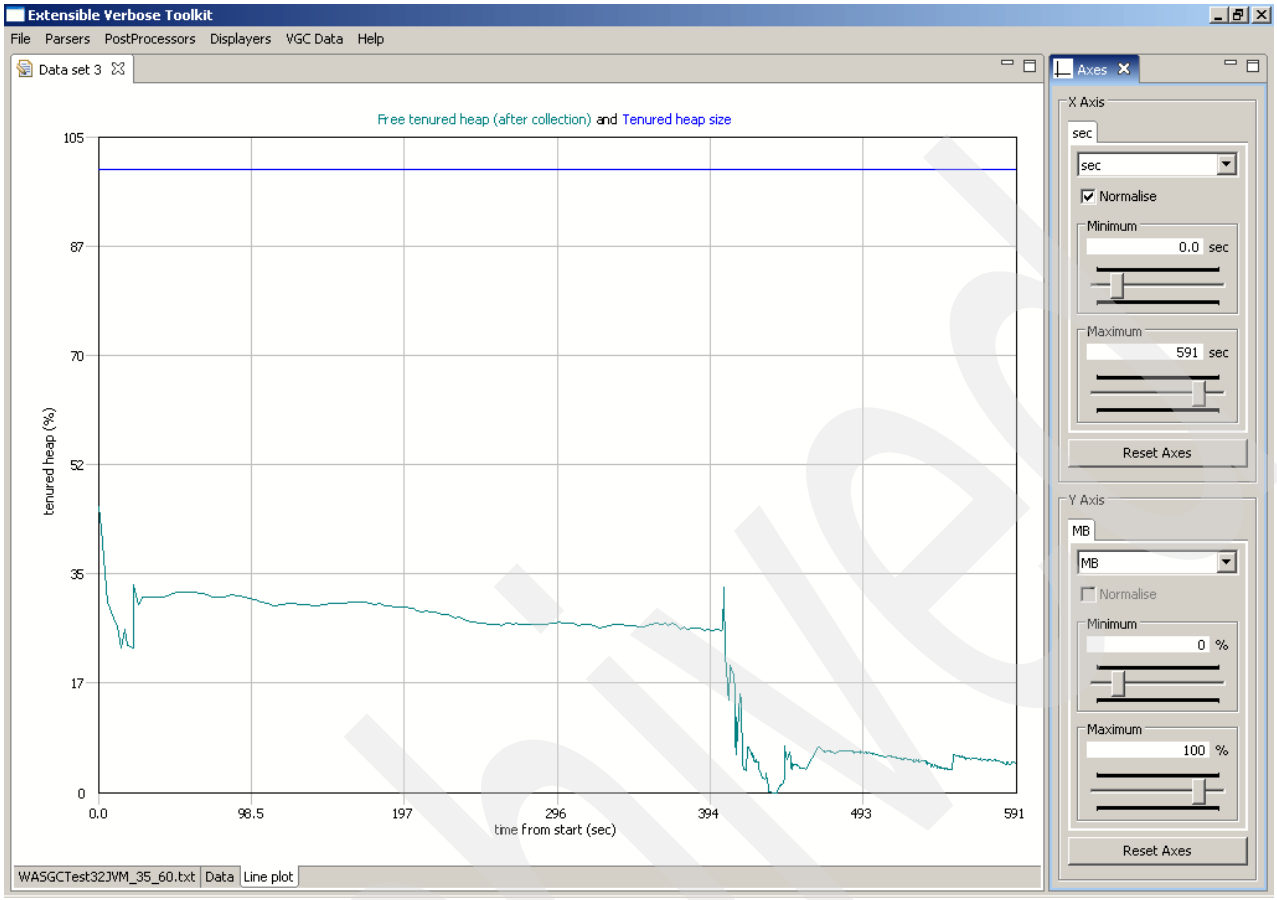


Figure 8-48 The default view displayed in EVTK after you open a verbose GC log file for analysis

The default graph did not meet our requirements. The data you want to visualize changes during the troubleshooting process, but fortunately displaying new graphs in EVTK is easy. You simply add or remove the data you want to plot through the VGC Data menu.

5. For instance, we wanted to look at how frequent and how long GC pauses were during the test period. To do this starting from the default graph shown in Figure 8-48:

   a. Select **VGC Data** then uncheck **Tenured heap size** from the menu, as shown in Figure 8-49.

      Notice EVTK dynamically updates the graph to reflect the selections in the VGC Data menu.

   b. Select **VGC Data** then uncheck **Free tenured heap (after collection)**.

      This gives a blank chart onto which you can plot the required data in the next step.

   c. Select **VGC Data** then check **Pause times (mark-sweep-compact) collections** to display GC pause data on the graph.

      This results in the chart shown in Figure 8-49.

See Figure 8-49.

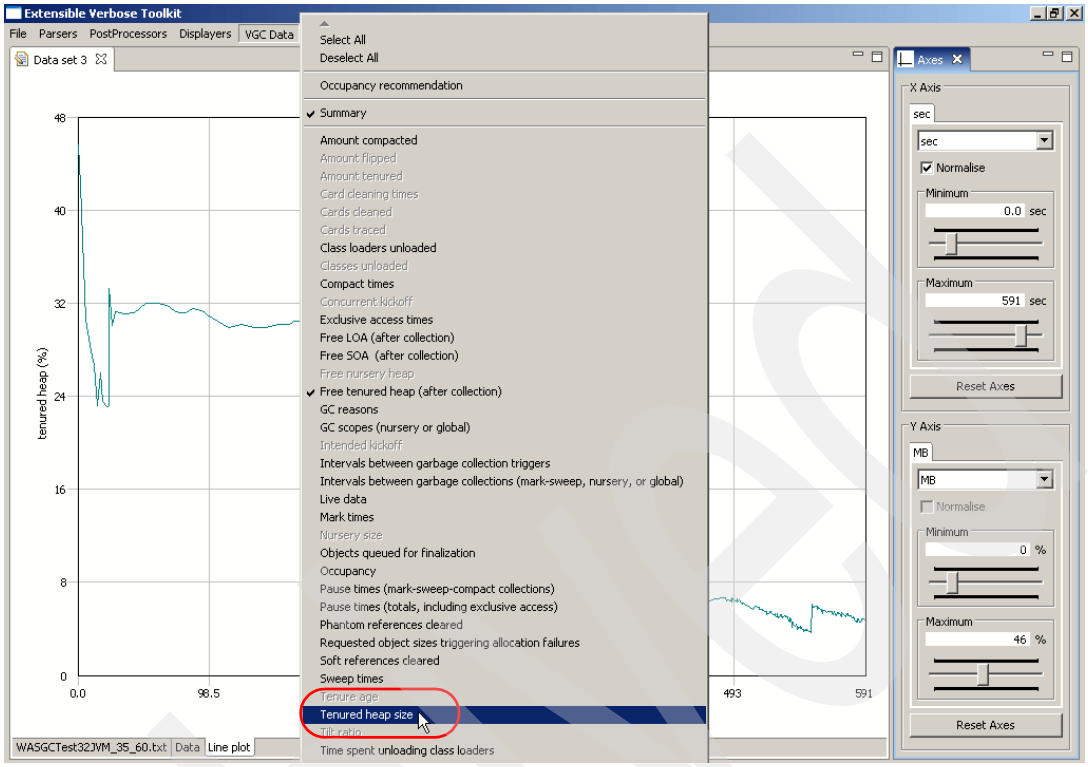


Figure 8-49 Deselect the Tenured heap size data to give a better scale for the free heap space data

Test results

The effect of insufficient space in the heap on application throughput, application response times, and GC pause times is clear from Figure 8-50, Figure 8-51, and Figure 8-52.

Figure 8-50 shows that average response times increase as the workload on the WebSphere Application Server increases.

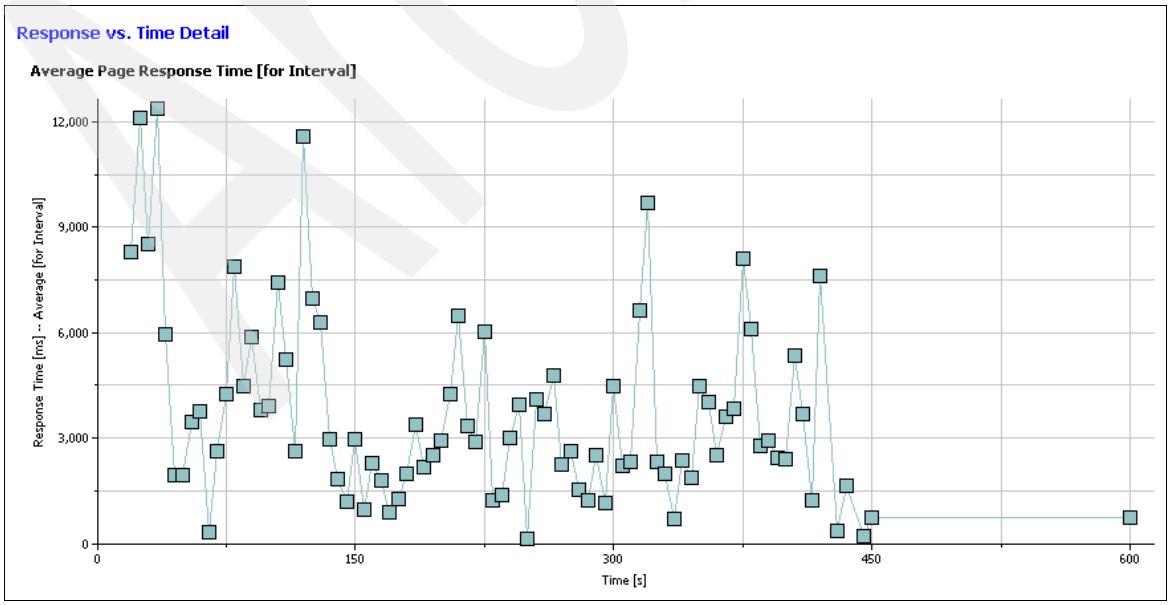


Figure 8-50 Response times increase as the workload increases

Note: The period of apparent response time decrease from approximately 400 seconds after test start onward, is misleading. The server at this time was occupied by the GC workload that tried to free up heap for new allocations. The response time in fact increased but this was not detected by Rational Performance Tester.

Figure 8-51 shows that the throughput of the Trade 6 application drops as the workload on the WebSphere Application Server increases, reaching a point of zero throughput when the garbage collector is running very frequently to ensure the heap does not completely run out of free space.

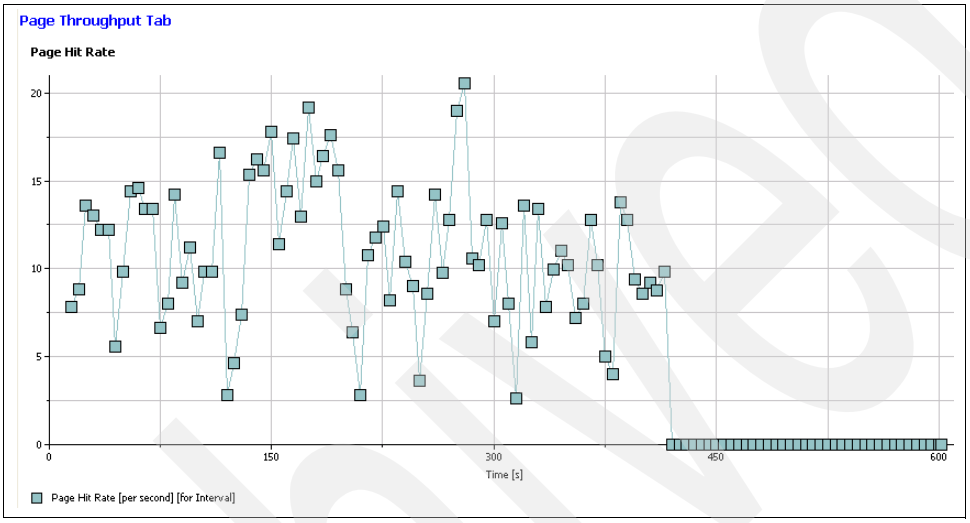


Figure 8-51 Throughput drops as response time (and GC pause times) increases

In Figure 8-52 you can see that as the workload increases, garbage collections occur more frequently, and take longer. This is because of the very low maximum heap size. As the workload increases, the garbage collector has to frequently compact the heap to prevent the heap running out of free space completely:

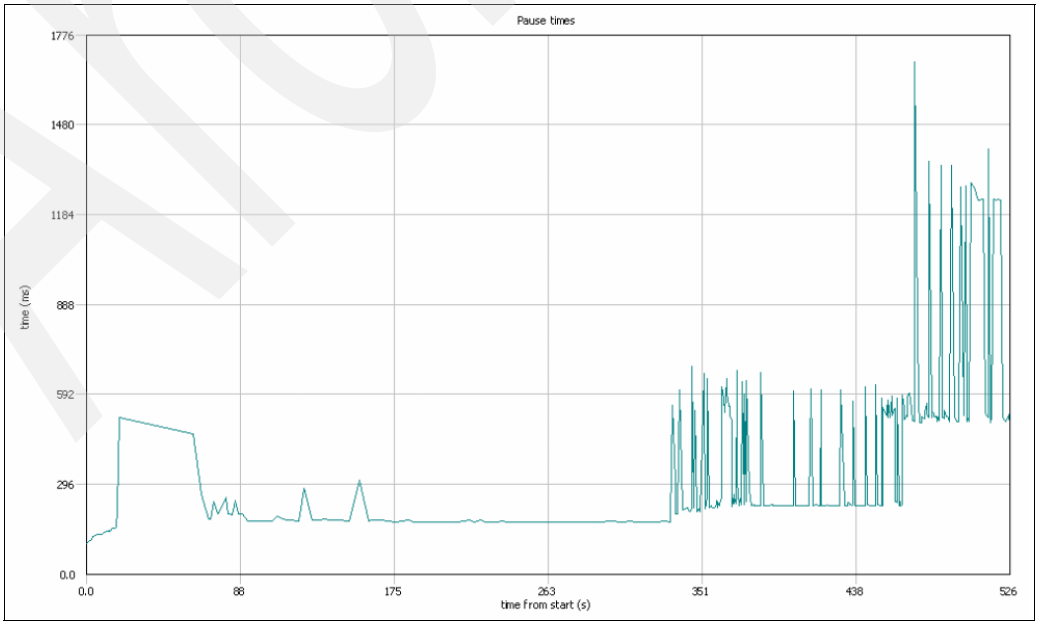


Figure 8-52 Garbage collection pauses increase in length and frequency as workload increases

Further analyzing verbose GC data with EVTK

EVTK provides several ways of displaying verbose GC data. For example, data can be exported in CSV format or displayed as a graph. You can chart a wide variety of GC data, featuring just one value plotted against elapsed time (such as free heap space) or combining multiple values on one chart. Typically this feature would be used to easily see the relation between multiple data values, such as free heap space and GC pause times. It is also possible to overlay data from multiple verbose GC logs on the same chart for comparative analysis between JVMs run with different GC settings.

In our case, we initially created a chart of the GC pause times in EVTK from the verbose GC data, using the steps in “Viewing verbose GC data in EVTK” on page 134. This clearly showed that GC pauses were occurring more frequently and getting longer as the workload increased. This lead us to suspect that the heap is running out of space under load.

You can use EVTK to easily verify your suspicion that the heap is running out of space. Re-plot the chart in EVTK to show the free heap space remaining against time. This gives the chart shown in Figure 8-53.

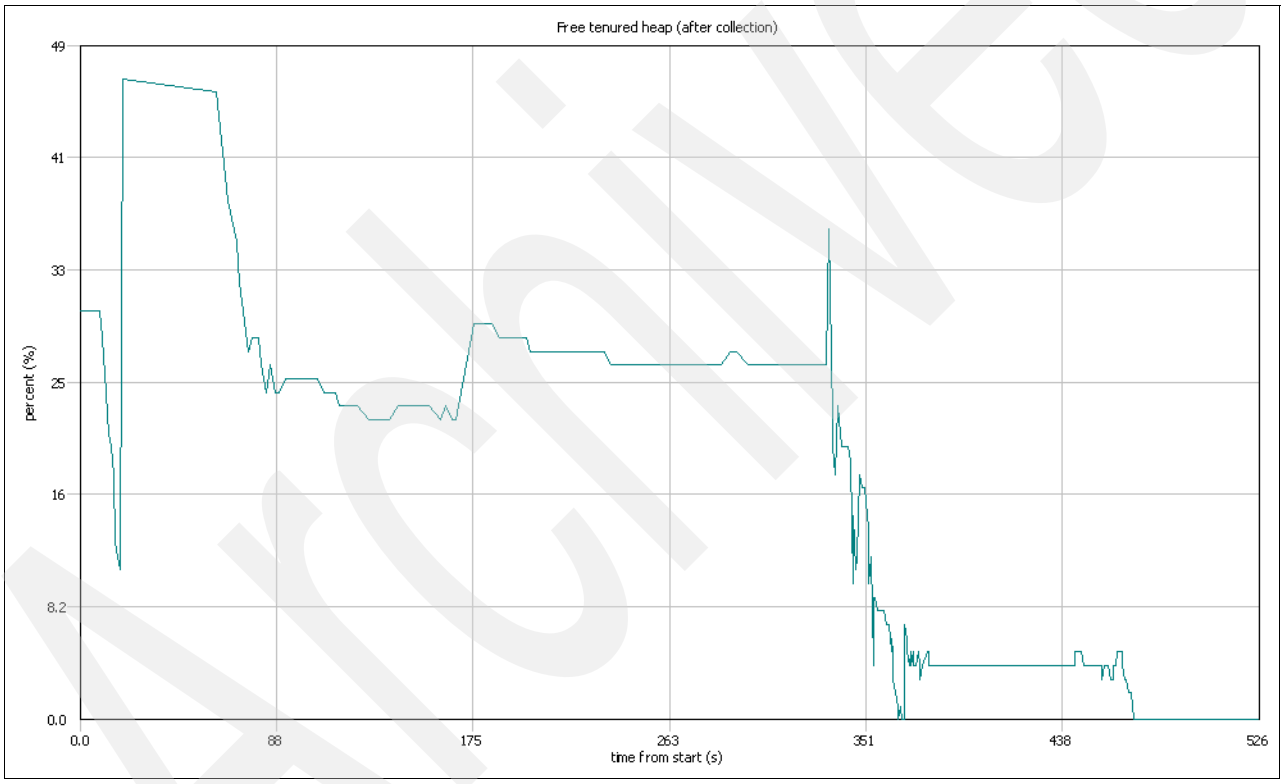


Figure 8-53 Plotting free heap size clearly shows that the heap is running out of space as workload increases

You can see that at the half-way point in the test, the available free heap space suddenly and sharply decreases to very low levels and the JVM eventually runs out of heap space.

You must, therefore, adjust the heap settings as follows:

```
-Xms50m -Xmx256m
```

Restart WebSphere Application Server so that the new heap size settings take effect.

Results with new heap size settings

Running the scenario again, the positive effect on application response time and throughput can be clearly seen in Figure 8-54 and Figure 8-55.

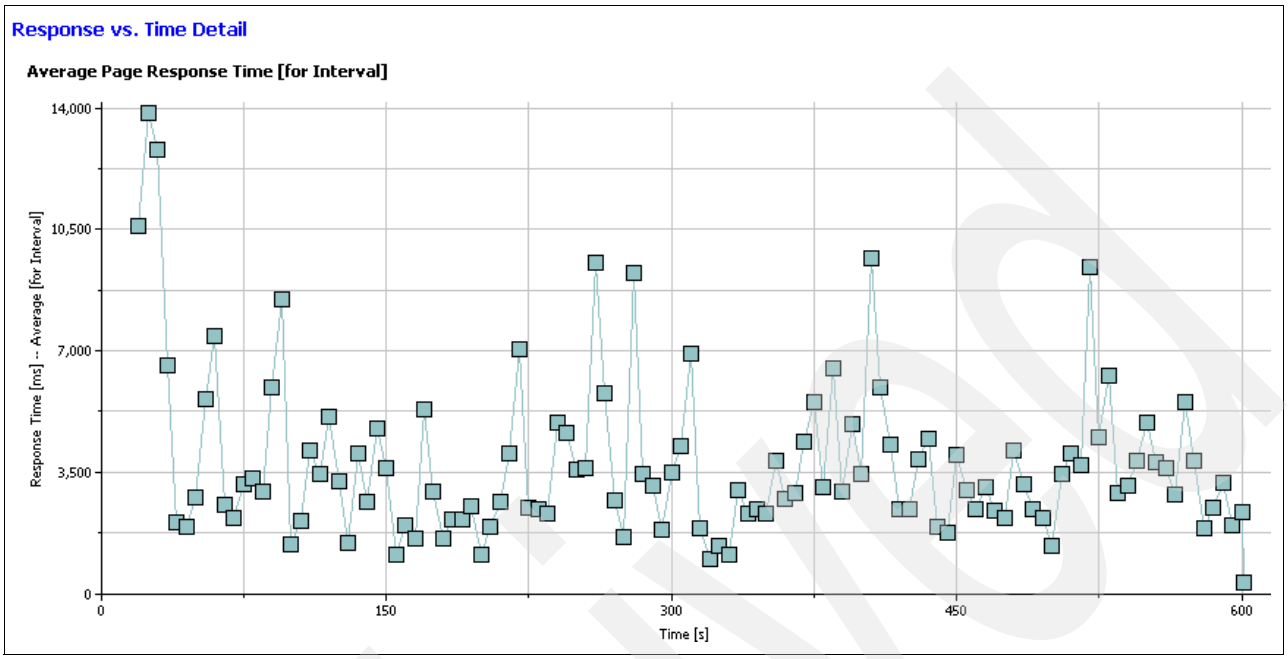


Figure 8-54 Average response time decreases after adjusting heap settings to more appropriate values

The application throughput also increases with the new heap settings, as shown in Figure 8-55.

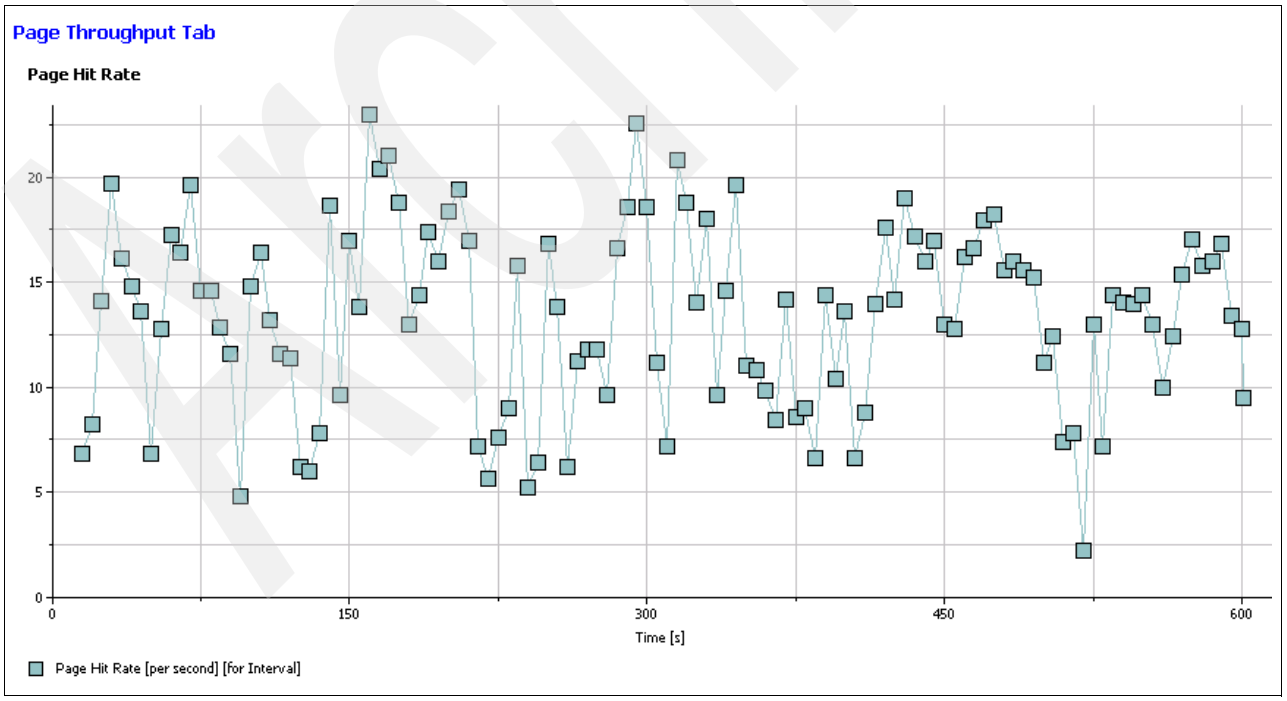


Figure 8-55 Application throughput increases after adjusting heap settings to more appropriate values

Confirming improved GC performance with EVTK

We collected verbose GC output from the second run of the application to compare to the previous analysis and confirm that GC is operating more efficiently.

EVTK allows you to plot data from multiple verbose GC logs simultaneously on the same chart. This is very useful for comparison purposes. In our scenario we overlaid the free heap data from the application run with new heap settings, over the data from the problematic run:

1. Copy the new verbose GC log file from i5/OS to the PC running EVTK.
2. Open the original verbose GC log file if it is not currently open and plot the “Free tenured heap (after collection)” data only:
   a. Select **File** → **Open File**.
   b. Select the original verbose GC log file and click **Open**. The default graph of the original data displays in EVTK showing Tenured heap size and Free tenured heap.
   c. Select **VGC Data**, then uncheck **Tenured heap size**.
3. Add the data from the latest test with new heap settings, to the graph:
   a. Select **File** → **Add File**.
   b. Select the new verbose GC log file and click **Open**.

The new data is then displayed as the blue plot line in the chart shown in Figure 8-56.

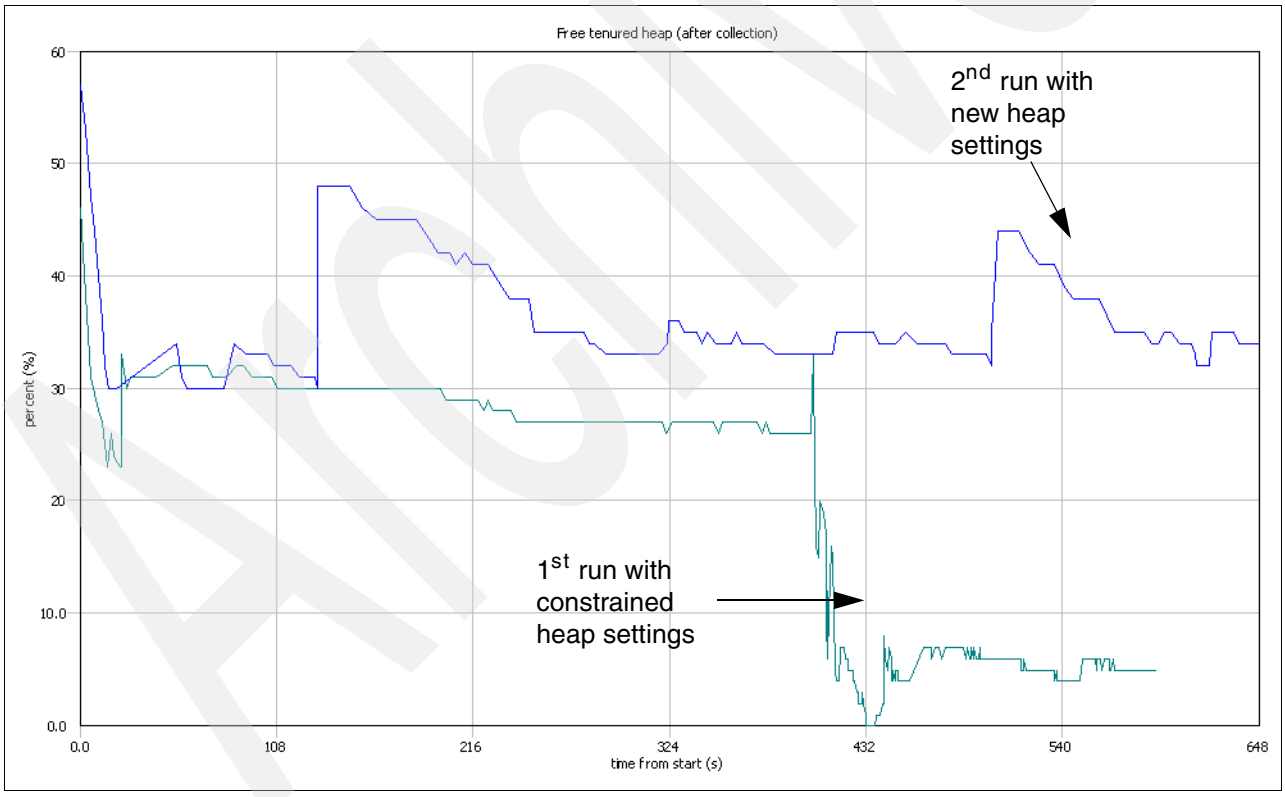


Figure 8-56 More free heap space is available after adjusting heap settings to more appropriate values

You can clearly see from the new data plot in Figure 8-56 that the heap no longer rapidly runs out of space as the application workload increases.

8.6.4 Application guidelines

Application design and coding techniques can have a major impact on performance. This section covers:

- Application coding tips and recommendations
- Application profiling and analysis: workstation
- Application profiling and analysis: server

Application coding tips and recommendations

One of the most common sources of application performance problems is excessive object creation. String concatenation is a prime example. Strings are easy to use, but are immutable (meaning every change involves another object creation). Each string object consumes working memory and increases the CPU usage during the garbage collection cycle.

If you have a lot of string manipulations in your program, for example, concatenating three or more strings, use StringBuffer as the target object for concatenated string. This is especially important if you do string concatenation in a loop.

For example, instead of using String to hold the result:

```
String s1 = "Use ";
....
s1 = s1 + "StringBuffer ";
.....
s1 = s1 + "for ";
...
s1 = s1 + "concatenation!";
System.out.println(s1);
```

Use the StringBuffer object:

```
StringBuffer sb = new StringBuffer("Use ");
...
sb.append("StringBuffer ");
...
sb.append("for ");
...
sb.append("concatination!");
System.out.println(sb.toString());
```

As the result, in the first code example 8 new objects have been created, while in the second example only 5. If you have to run this code segment as part of the loop you would create `3 x <number of iterations>` less objects with the second approach.

Another source of application performance problems is using "expensive" tasks such as synchronized methods and reflection. They can be very useful, just bear in mind that they tend to be fairly resource intensive. Another typical error is in overusing exception blocks. Use exception blocks for actual "exceptions", not for controlling the application flow.

Application profiling and analysis: workstation

It is recommended thoroughly testing and profiling your application prior to putting it into production under IBM Technology for JVM for i5/OS. However, it might be the case that performance issues might not surface until the application is in production and placed under heavy load. In this case it might be necessary to perform profiling activities to establish where any bottlenecks might be occurring in the application.

IBM JVMs prior to JDK version 5.0 implemented the Java Virtual Machine Profiling Interface (JVMPI). JVMPI provides an interface between the JVM and a profiling agent. IBM provides a Remote Agent Controller (RAC) application as the profiling agent for its JVMs. The RAC is available on multiple platforms, including i5/OS. When the JVM is started in profile mode, it links with the RAC and notifies it of events such as method calls and memory allocation. The RAC issues control requests through the JVMPI, such as what events to monitor.

The JVMPI has been deprecated in JDK version 5.0 in favor of the Java Virtual Machine Tools Interface (JVMTI). JVMTI is the strategic interface for application monitoring and performance analysis.

IBM JVMs use a remote agent controller server daemon to implement JVMPI and JVMTI. These are available on a variety of platforms (including System i platform) and are currently packaged with application development tools such as WebSphere Development Studio Client for iSeries. There are two key client application interfaces to JVMPI and JVMTI:

- Tivoli Performance Viewer, which is integrated within the WebSphere Application Server administration console
- Rational Software Development Platform profiling and logging perspective

Tivoli Performance Viewer provides basic information about response time for servlets, JavaServer™ Pages™ (JSP™), and Enterprise JavaBeans™ (EJB™) methods, and also detailed information about JVM statistics such as garbage collection. Refer to Appendix A, “Running WebSphere Application Server with IBM Technology for JVM” on page 167 for more details.

WebSphere Development Studio Client for iSeries Advanced Edition provides detailed information about method execution time, memory usage, thread analysis and code usage in J2EE and Java 2 Standard Edition (J2SE) applications. The next few sections illustrate how to identify potential performance and functional problems in Java applications using WebSphere Development Studio Client for iSeries.

Attention: At the time of this writing, WebSphere Development Studio Client for iSeries was using JVMPI interface. However, the concepts must still apply to JVMTI when the appropriate support is available in the tool.

Analyzing response time and memory usage

This example shows how you can use WebSphere Development Studio Client for iSeries Advanced Edition to analyze an application’s response time components and determine memory and object usage.

Restriction: At the time of this writing, only the Classic JVM could be used for remote debugging in WebSphere Development Studio Client for iSeries version 6.0.1. However, the approach presented in this section is applicable to IBM Technology for JVM when this is supported for remote debugging.

Perform the following steps:

1. Start WebSphere Development Studio Client for iSeries Advanced Edition on your workstation.
2. Open the Java perspective.
3. Locate the Java application project and source files for your application.

4. Right mouse click the class (which must have a main() method) and select **Profile** → **Java Application**.
5. At the properties window select the **Profiling** tab. See Figure 8-57.

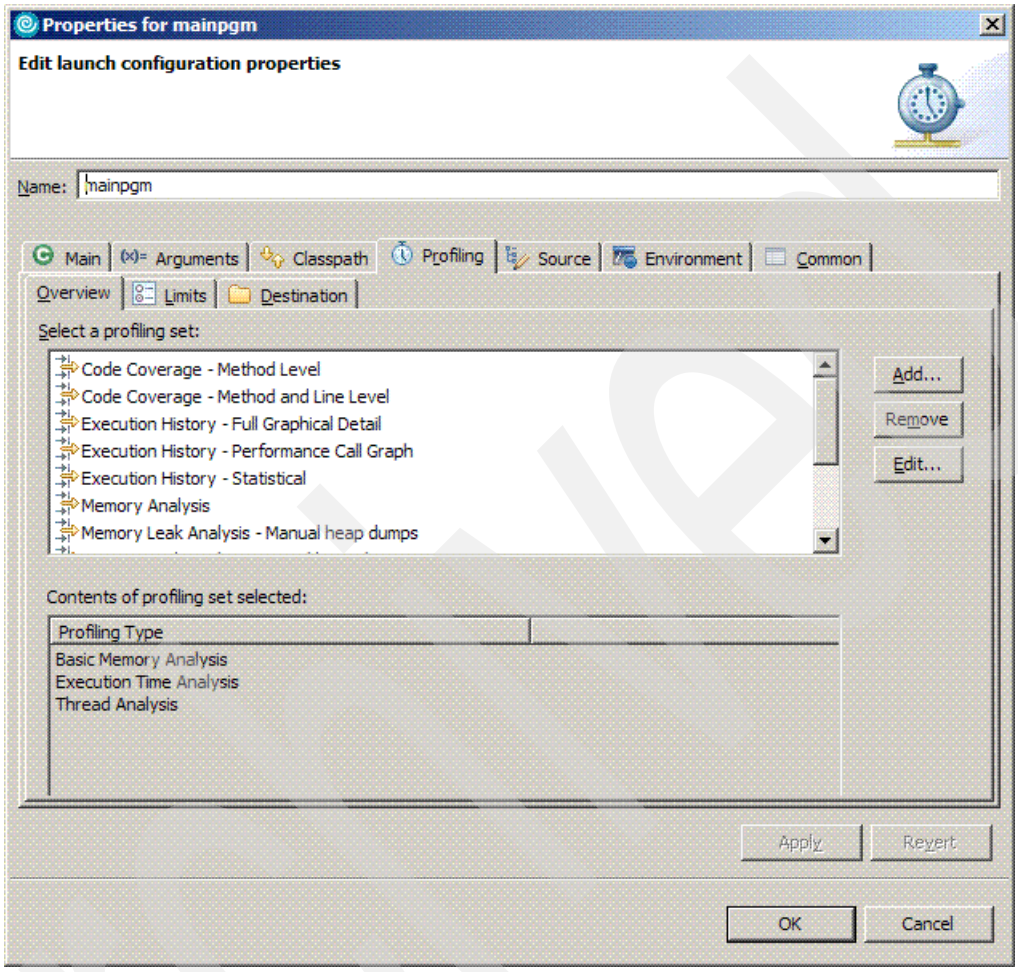


Figure 8-57 Application profiling selections

6. Select the **Basic Memory Analysis** option and click **Edit**.
7. At the Edit Profiling Set window enter a descriptive profiling set name and description. Click **Next**.
8. Expand the Time Analysis section and select the **Execution Time Analysis** option.
9. Select the **Collect method CPU time** information option.

See Figure 8-58.

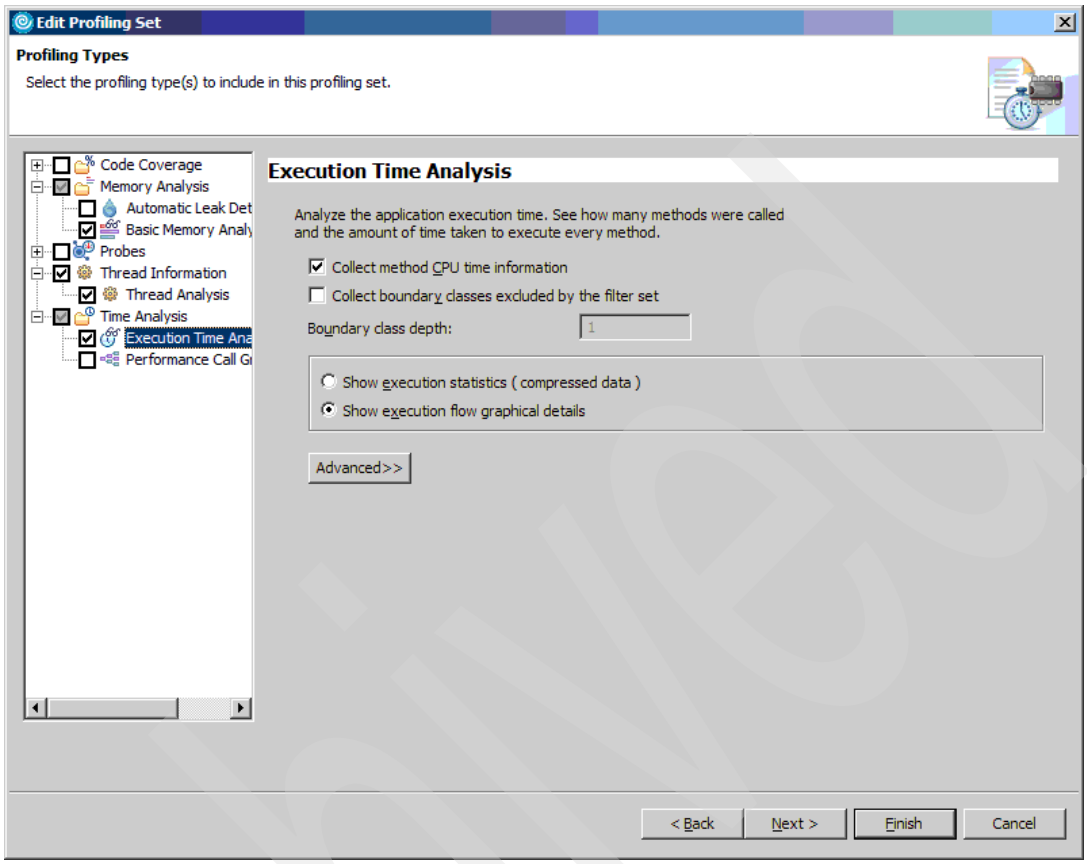


Figure 8-58 Monitor basic memory usage and method response times

10. Click **Next** and you must see the default filter set selected.
11. In the *Contents of selected filter set* section click the **Add** button.
12. Specify the package you want to monitor. You can use the wildcard character * to select all classes in the package.

    By default, all methods are selected and the filter rule is included.

    Click **OK** when you are finished.

See Figure 8-59.

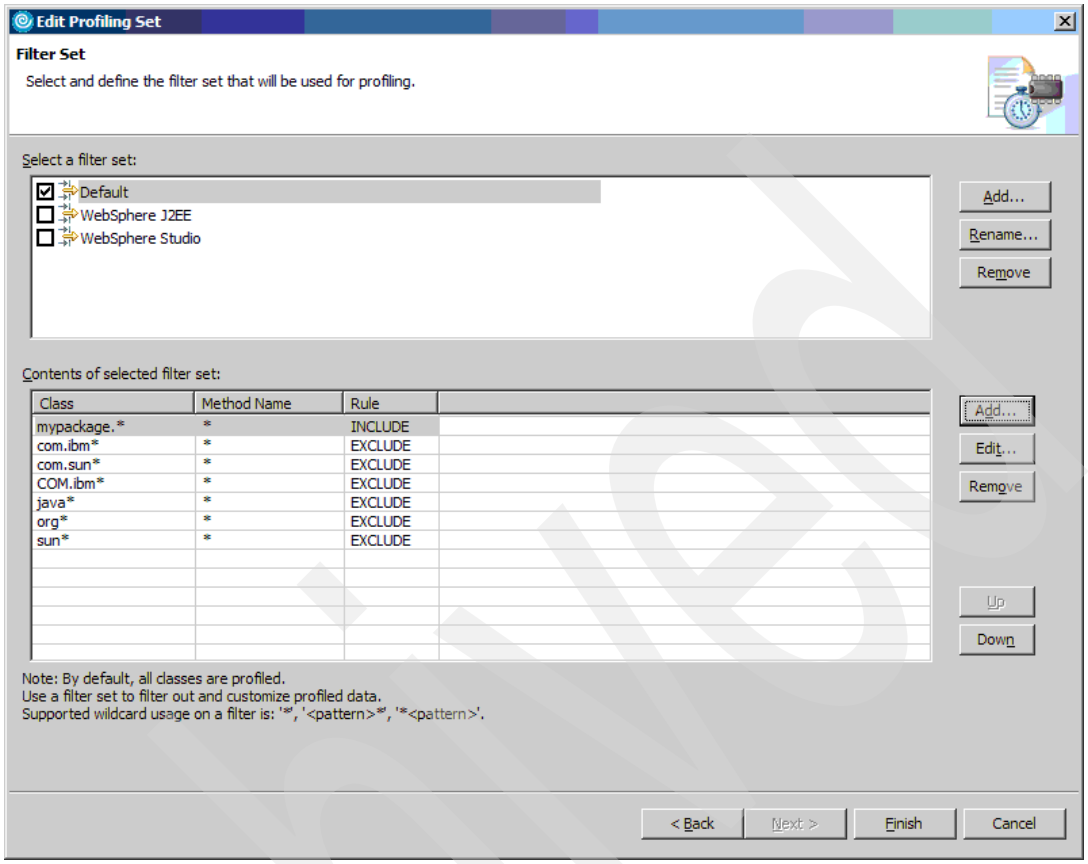


Figure 8-59 Specify the classes and methods to monitor

13. You can repeat the previous steps to monitor any additional packages. After you have specified all classes to profile click **Finish**.
14. You must be back to the properties window, as shown in Figure 8-57 on page 143. Click **OK** to profile the application.
15. If you do not have the profiling and logging perspective open, you see a “confirm perspective switch?” window open. If so, click **Yes**.
16. You will be in the profiling and logging perspective and see your application status as monitoring. At the toolbar select **Window** → **Show View** → **Console**. This displays any standard out data.
17. After your application completes you must see your application status reported as terminated. Verify that all output data is correct, because you want to ensure the application ran properly.
18. Expand the terminated link and you see options for:
    - Basic Memory Analysis
    - Execution Time Analysis
    - Thread analysis

19. Right mouse click the Basic Memory Analysis option and select **Open With** → **Memory Statistics**. See Figure 8-60.

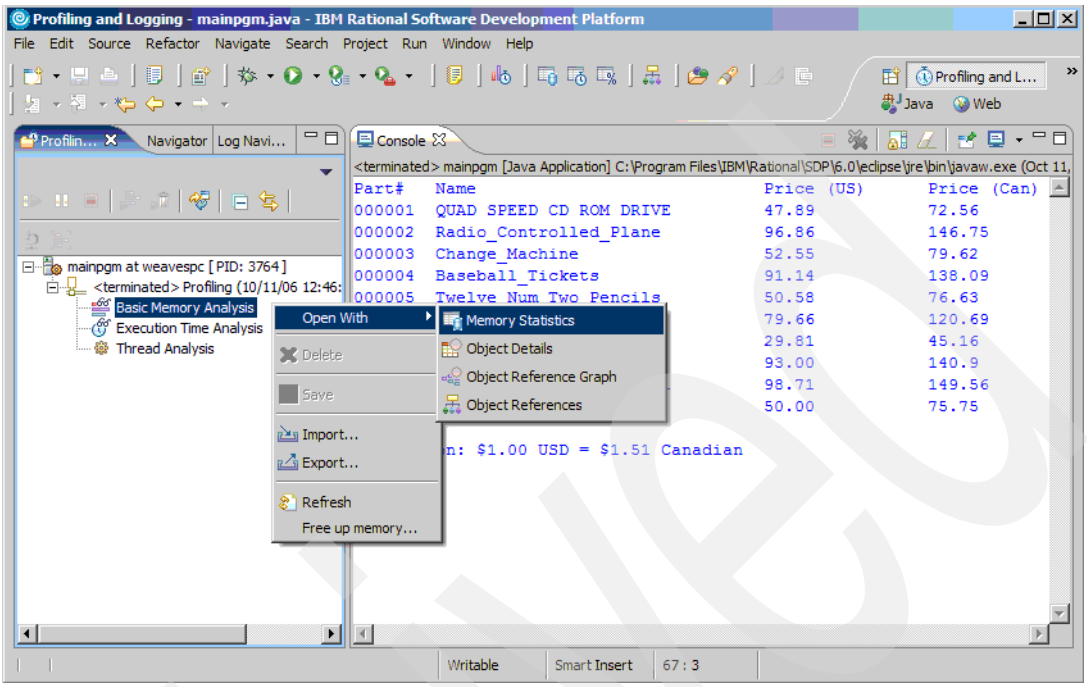


Figure 8-60 Gather basic memory usage statistics in the profiling and logging perspective

20. You see statistics for memory and object usage, sorted by application package. You must see these metrics, as shown in Figure 8-61:

   – Total Instances

     The total number of instances that have been created of the selected package, class, or method.

   – Live Instances

     The number of instances of the selected package, class, or method where no garbage collection has taken place.

   – Collected

     The number of instances of the selected package, class, or method that were removed during garbage collection.

   – Total Size

     The total size (in bytes) of the selected package, class, or method of all instances that were created for it, including objects that were garbage collected.

   – Active Size

     The summed size of all live instances.

See Figure 8-61.

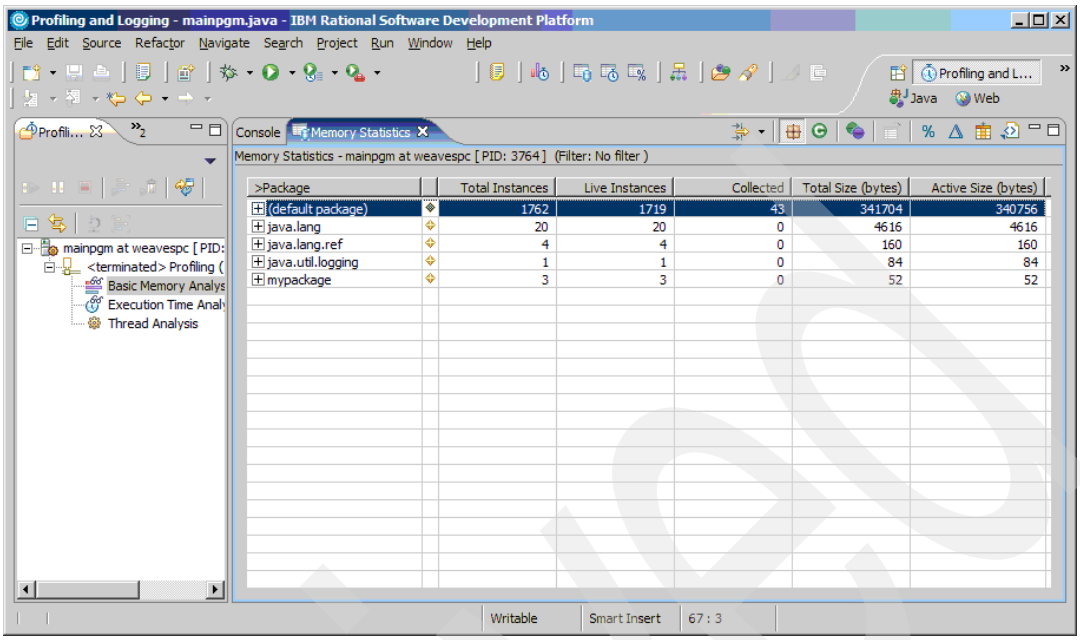


Figure 8-61 View basic memory usage statistics in the profiling and logging perspective

In this example you can see that the default package, which is where the test application resides, generates the most activity. If you expand the default package you can see statistics for primitives such as character, byte, integer, and so on.

If the application is still active, you can continue running it and analyze the number of objects and memory usage. The profiling and logging perspective has icons which show increases or decreases. You can use this technique to help detect excessive object creation or a memory leak. See Figure 8-62.

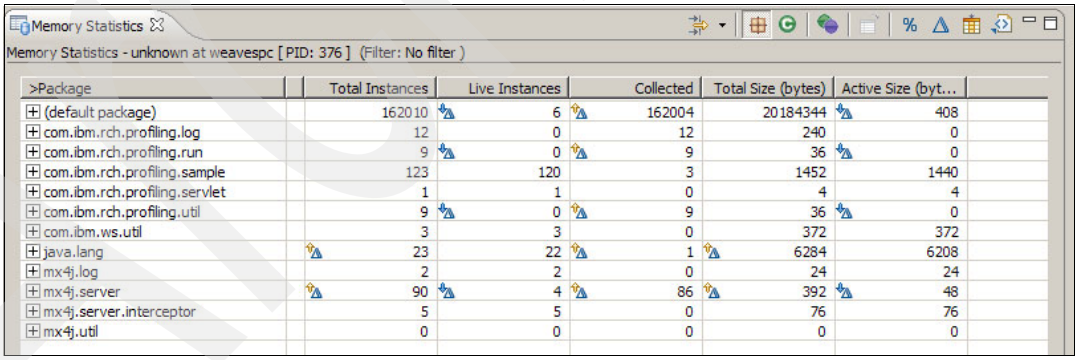


Figure 8-62 Memory statistics view shows increases and decreases in objects

21. The next step is to analyze response time components. At the profiling monitor view right-click the execution time analysis option and select **Open With → Execution Statistics**. See Figure 8-63.

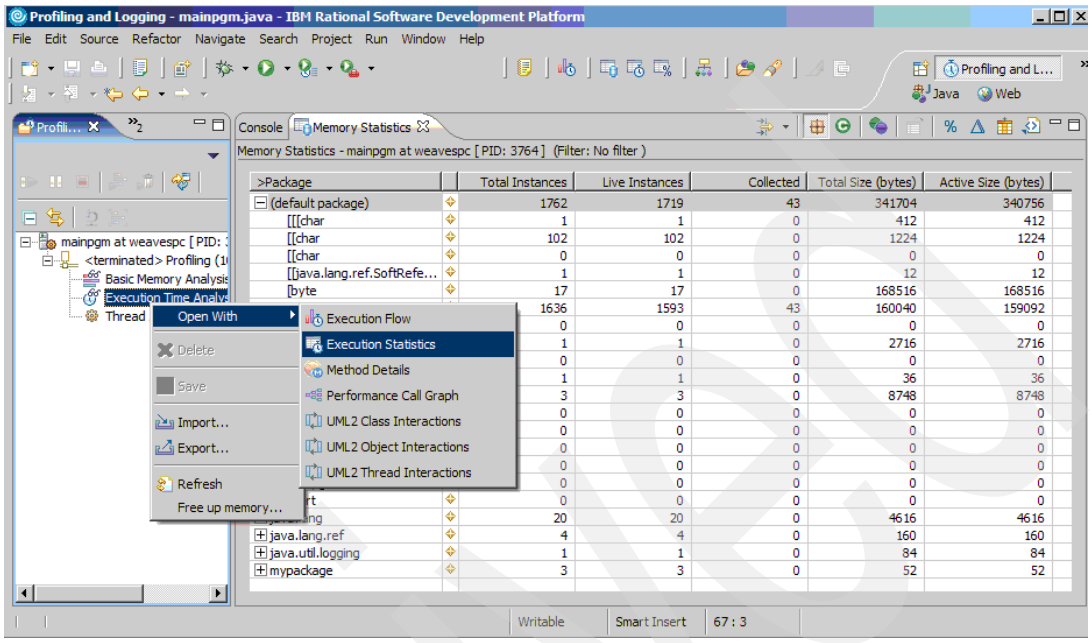


Figure 8-63 Gather basic response time statistics in the profiling and logging perspective

22. You must see a view with response times sorted by application package. See Figure 8-64. You can then expand the packages and the response times for individual methods. You must see the following metrics:

   – Base Time

     This is the time taken to execute the invocation, excluding the time spent in other methods that were called during the invocation. This is typically the most useful metric.

   – Average Base Time

     The base time divided by the number of calls.

   – Cumulative Time

     This is the time taken to execute all methods called from an invocation. If an invocation has no additional method calls, then the cumulative time is equal to the base time.

   – Calls

     The number of calls made by a selected method.

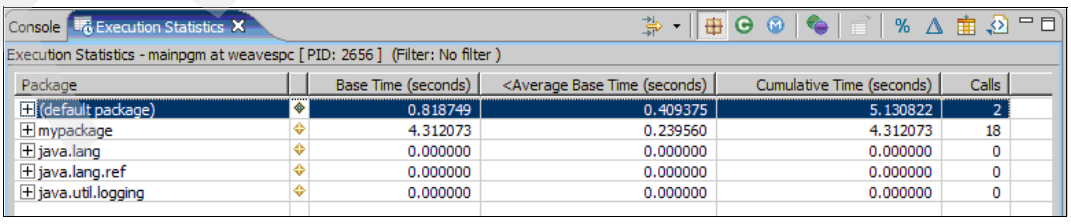
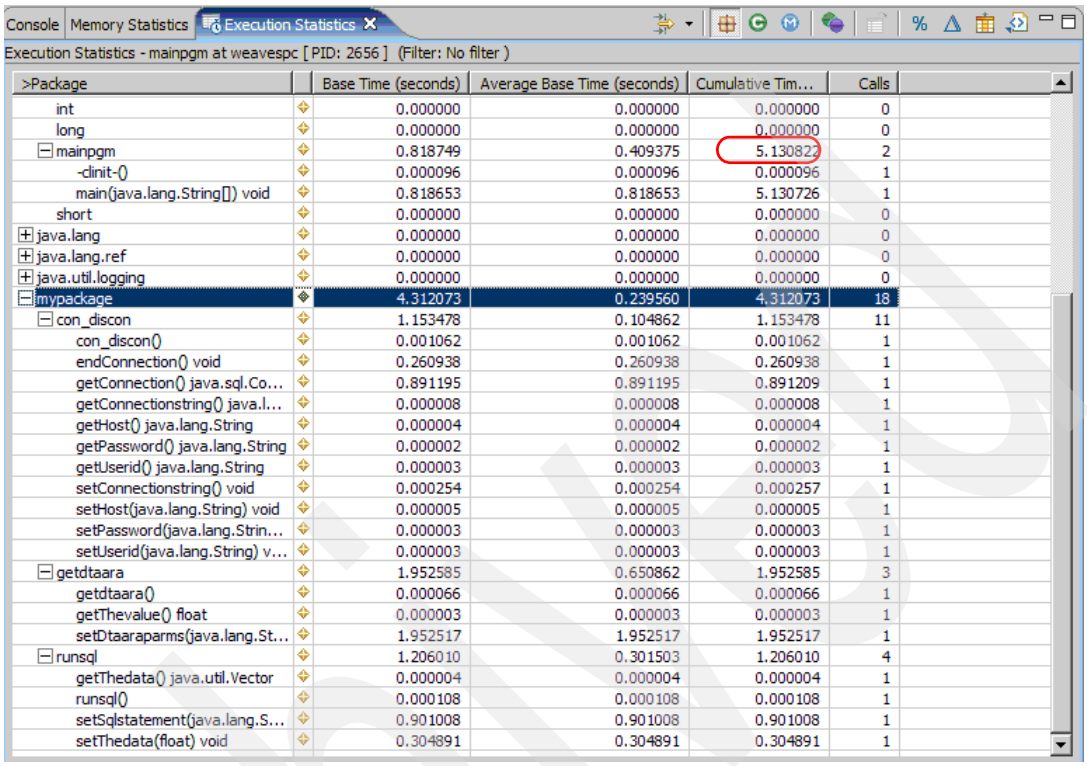


Figure 8-64 Response times sorted by package

In this example you can see that the mypackage package has the largest contribution to response time.

23. If you expand the packages you can view response times for individual methods. See Figure 8-65. You can click the **Base Time (seconds)** column heading (or any other heading) to sort in ascending or descending order.



>Package	Base Time (seconds)	Average Base Time (seconds)	Cumulative Tim...	Calls
int	0.000000	0.000000	0.000000	0
long	0.000000	0.000000	0.000000	0
mainpgm	0.818749	0.409375	5.130822	2
-clinit-()	0.000096	0.000096	0.000096	1
main(java.lang.String[]) void	0.818653	0.818653	5.130726	1
short	0.000000	0.000000	0.000000	0
java.lang	0.000000	0.000000	0.000000	0
java.lang.ref	0.000000	0.000000	0.000000	0
java.util.logging	0.000000	0.000000	0.000000	0
mypackage	4.312073	0.239560	4.312073	18
con_discon	1.153478	0.104862	1.153478	11
con_discon()	0.001062	0.001062	0.001062	1
endConnection() void	0.260938	0.260938	0.260938	1
getConnection() java.sql.Co...	0.891195	0.891195	0.891209	1
getConnectionstring() java.l...	0.000008	0.000008	0.000008	1
getHost() java.lang.String	0.000004	0.000004	0.000004	1
getPassword() java.lang.String	0.000002	0.000002	0.000002	1
getUserid() java.lang.String	0.000003	0.000003	0.000003	1
setConnectionstring() void	0.000254	0.000254	0.000257	1
setHost(java.lang.String) void	0.000005	0.000005	0.000005	1
setPassword(java.lang.Strin...	0.000003	0.000003	0.000003	1
setUserid(java.lang.String) v...	0.000003	0.000003	0.000003	1
getdtaara	1.952585	0.650862	1.952585	3
getdtaara()	0.000066	0.000066	0.000066	1
getThevalue() float	0.000003	0.000003	0.000003	1
setDtaaraparms(java.lang.St...	1.952517	1.952517	1.952517	1
runsql	1.206010	0.301503	1.206010	4
getThedata() java.util.Vector	0.000004	0.000004	0.000004	1
runsql()	0.000108	0.000108	0.000108	1
setSqlstatement(java.lang.S...	0.901008	0.901008	0.901008	1
setThedata(float) void	0.304891	0.304891	0.304891	1

Figure 8-65 Detailed execution time for each method

In this example you can see the following:

- Total execution time is 5.13 seconds (cumulative time for the mainpgm class) as shown in Figure 8-65.
- The main method in the mainpgm class execution time is .82 seconds (base time).
- The con_discon class execution time is 1.15 seconds, and the getConnection method is the largest component.
- The getdtaara class execution time is 1.95 seconds and the getDtaaraparms is the largest component.
- The runsql class execution time is 1.21 seconds and the setSqlstatement is the largest component.

You can right-click any row and see options for viewing the source code or method invocation.

24. The next step is to use the performance call graph to view the application execution environment. At the profiling monitor view right-click the execution time analysis option and select **Open With → Performance Call Graph.**

The performance call graph shows the threads and application method's in a hierarchical relationship.

See Figure 8-66.

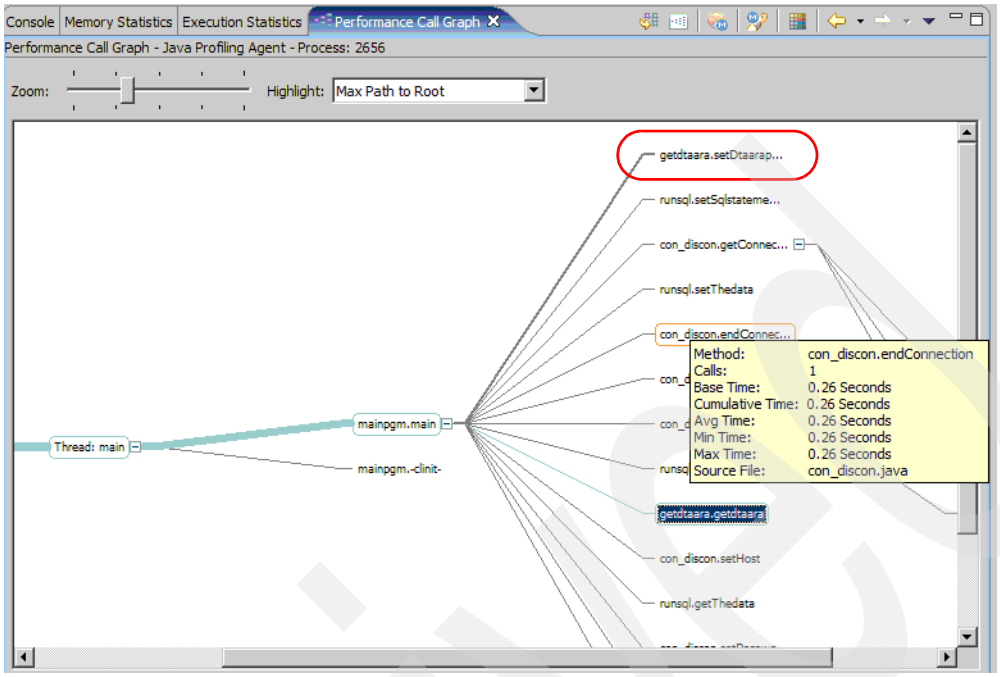


Figure 8-66 Performance call graph example

In this example you can see that the most lengthy method (getdtaara.setDtaaraparms()) has the thickest line. You can also see that the application is executed in a single thread. If you hover the mouse cursor over the individual methods, you see a pop-up window that lists the response time metrics and source file.

Application profiling and analysis: server

It is most beneficial to do application profiling and analysis during development and testing phases, rather than in the production phase. It is also easier to do profiling and analysis with the Rational Software Development Platform tools, such as WebSphere Development Studio Client for iSeries Advanced Edition, rather than server-based tools such as Performance Explorer (PEX) in i5/OS. However, you might find it beneficial to analyze the application running on System i platform. For example, PEX shows detailed CPU usage at the job and thread level. PEX is also beneficial for analyzing application performance issues when the server is running with production level workloads.

This example illustrates creating a Performance Explorer (PEX) trace for a Java application that utilizes IBM Technology for JVM, then interpreting the results with Performance Trace Data Visualizer (PTDV). You can see how to determine the CPU time for the JVM process, and also the jobs that access i5/OS resources via the IBM Toolbox for Java. You can also see how to determine the relative impact of garbage collection. PTDV can be downloaded from the IBM alphaWorks® Web site at:

`http://www.alphaworks.ibm.com/tech/ptdv`

Note: i5/OS provides the support for creating a Performance Explorer trace. You must install the Performance Tools for iSeries product (5722-PT1) if you want to print reports.

Perform the following steps:

1. Start a 5250 emulator session and sign in.
2. Enter the following command to create a PEX definition:

   ```
   ADDPEXDFN DFN(TPROF5MS) TYPE(*TRACE) JOB((*ALL)) MAXSTG(1000000) INTERVAL(5)
   TRCTYPE(*SLTEVT) SLTEVT(*YES) BASEVT((*PMCO *NONE *FORMAT2)) TEXT('Tprof with
   sampling interval of 5 ms')
   ```

3. This command creates a trace that samples every 5 milliseconds and records specific events. Next, start the trace by issuing the following command (give your trace a good descriptive name):

   ```
   STRPEX SSNID(JavaTprof) DFN(TPROF5MS)
   ```

   Notice that you want to use the PEX definition created in the previous step: `TPROF5MS`.
4. Run your Java application.
5. Enter the command:

   ```
   ENDPEX SSNID(*SELECT)
   ```

6. The ENDPEX screen shows how many events have been collected. Press F5 to refresh the event count. For Java applications 100,000 to 500,000 events usually provide a good sample. After enough events have been collected, select your collection and press Enter.
7. Start PTDV on your workstation.
8. You must see the load PEX data window open as shown in Figure 8-67.

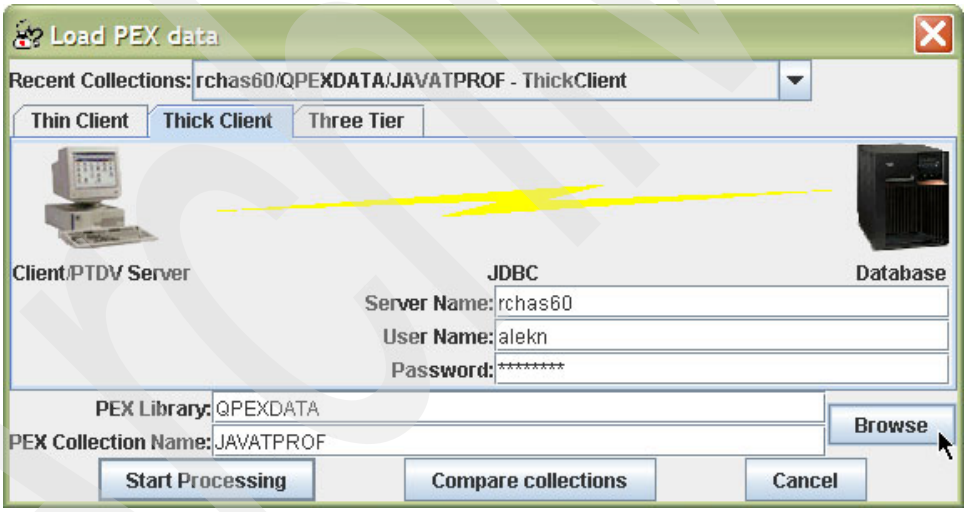


Figure 8-67 Specify the PEX data to import

9. Select the **Thick Client** tab and enter the server name, user name and password.
10. You can click the **Browse** button to search for your PEX collection. Otherwise, enter the library and collection name manually.
11. Click the **Start Processing** button to start the analysis.
12. You must see a TPROF options window open. Accept the defaults and click **OK**.

    You must be at the cumulative information tab, which shows information about when the trace was taken, the number of events recorded, sampling interval, and sample time. You must also see system information such as the system model, amount of processors, number of disk arms, and other data.

13. Click the **Job/Thread List** tab. You must see a list of events causing jobs and metrics such as the number of events and CPU usage. Click the **Cum CPU Time (ns)** column to sort it in descending order.

14. Figure 8-68 shows the jobs consuming the most CPU. In this example the WebSphere job (TEST32JVM) consumes most of the CPU amongst all jobs (the '//' job represents the SLIC threads activity and we do not count it).

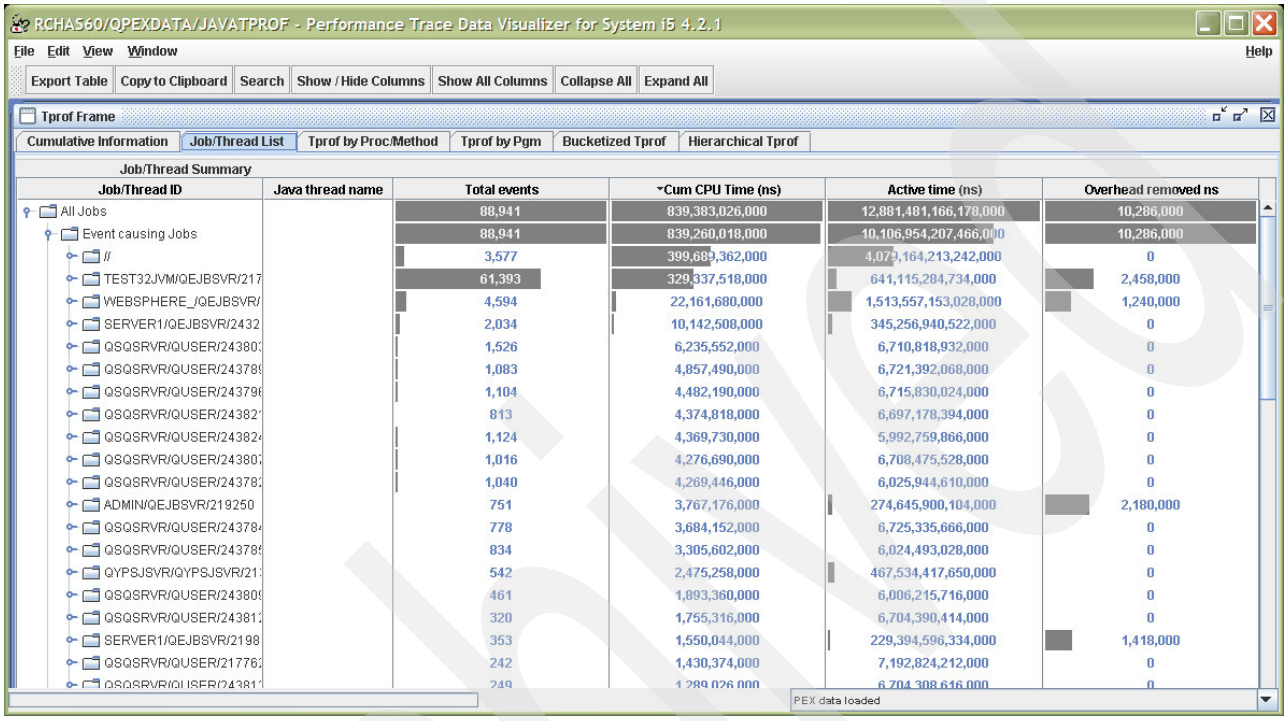


Figure 8-68 Job listing sorted by cumulative CPU time

The application performs a database query, using the Native JDBC driver. Notice that the associated QSQSRVR jobs consumed a fairly small amount of CPU.

15. Expand the job link and you see the job's threads and the associated number of events and CPU usage, as shown in Figure 8-69.

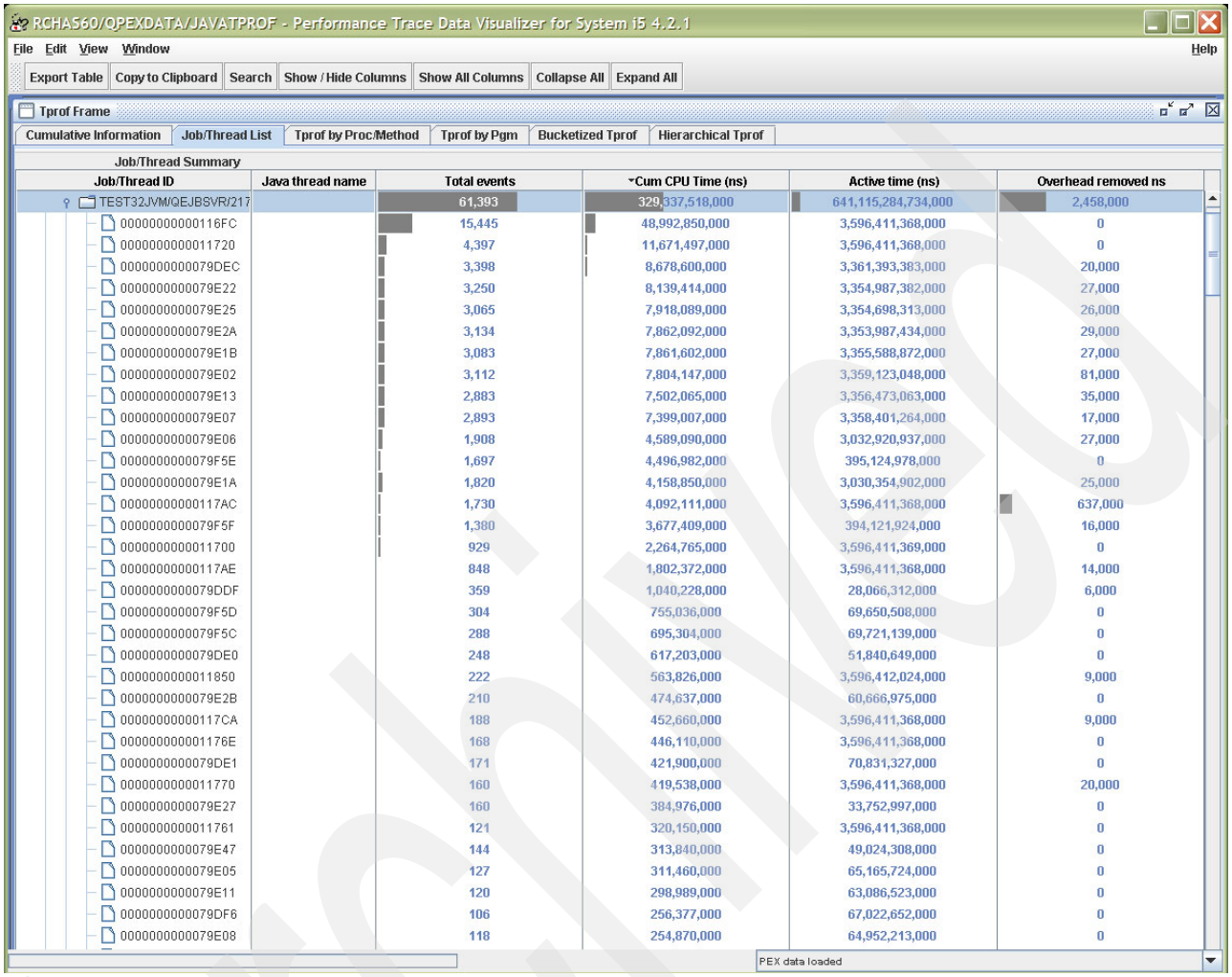


Figure 8-69 View events and CPU time by job thread

16. The next step is to view the details for the **TEST32JVM** job. Double-click it.

17. Click the **Bucketized Tprof** tab to get a more user friendly view of the data, as shown in Figure 8-70.

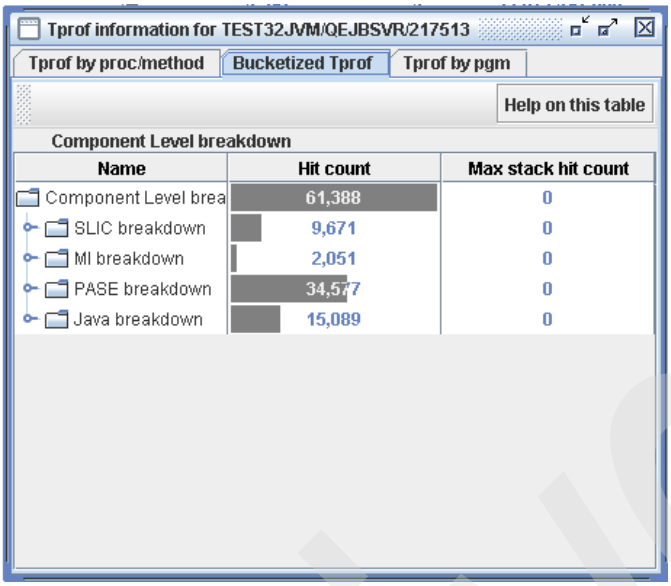


Figure 8-70 View the event count by category

18. You must see a component level breakdown view for the number of hits to SLIC (System i microcode), MI (i5/OS tasks), PASE (where IBM Technology for JVM runs), and JAVA.

    Note that the PASE component has the most hits in this example. This is because the applications were run with IBM Technology for JVM, and there was very little other workload running during the trace. If they had been run with Classic JVM the hits would be displayed in the SLIC and Java breakdown categories.

19. Expand the PASE breakdown category. See Figure 8-71.

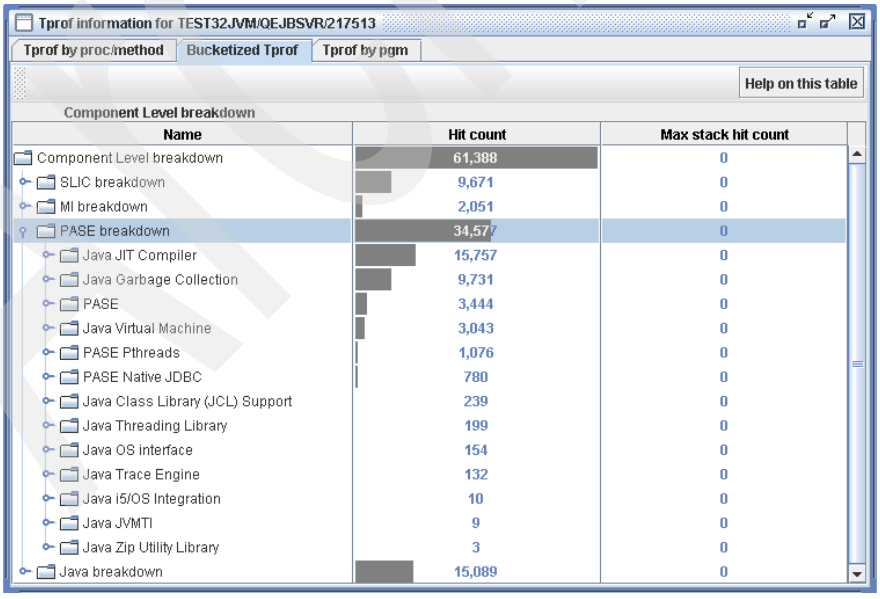


Figure 8-71 View the activity within the PASE

20. The PASE breakdown shows the relative activity for the IBM Technology for JVM components.

8.7 Application debugging

In most situations the Java application developer uses workstation-based tools to build and test their applications prior to deployment on System i platform. However, there might be situations when the application is deployed to System i platform and it does not function properly, especially in production environments under heavy load. In this section you learn how you can debug a Java application running on System i platform.

8.7.1 Review any available data

Your first step must be to verify that the application has in fact run properly in a test environment (on a workstation or on System i platform). Assuming that it has, check any available job log, spool file, or JVM console information for clues. For example, your application might be using the IBM Toolbox for Java classes. The password for the i5/OS user profile that it is using might be expired. If so, you would see an authentication error message to that effect in the JVM console. Also, other issues such as network problems might be causing functional problems in the application.

8.7.2 Debugging a Java application running on System i platform with WebSphere Development Studio Client for iSeries

WebSphere Development Studio Client for iSeries provides a complete development environment for traditional interactive and batch applications, Web applications, and Java applications. It includes local test environments for testing and debugging your stand-alone, and WebSphere Application Server based applications. However, you have to test your application on a server (System i platform in our case) before deploying it in a production environment.

WebSphere Development Studio Client for iSeries has a comprehensive debugging environment for your Java applications. It allows you to set break points in your code, monitor variables, view thread status, and many other tasks. WebSphere Development Studio Client for iSeries allows you to do Java development on your workstation and you can test and debug your applications locally (the application runs in a JVM on your workstation) or remotely (the application runs in a JVM on System i platform).

Restriction: At the time of this writing, only the Classic JVM can be used for remote debugging in WebSphere Development Studio Client for iSeries version 6.0.1. However the approach presented in this section is applicable to IBM Technology for JVM when this support is available.

The following are the basic steps you have to follow to debug a Java application remotely on a System i platform:

1. Create or import a Java application project in your WebSphere Development Studio Client for iSeries workspace.
2. Create, import, or modify the Java source files for your application.
3. Specify the appropriate breakpoints in your application code.
4. Test the application locally to ensure it actually works.
5. Define a remote system explorer connection to the System i platform.

6. Copy your application to the System i file system.
7. Start a debug session to run the application on the System i platform.
8. Step through the application and monitor the variables, thread status, and output data.

Steps 1 and 2 are not covered here. If necessary, refer to *Rational Application Developer V6 Programming Guide*, SG24-6449.

Specify appropriate breakpoints in application code

Setting breakpoints in WebSphere Development Studio Client for iSeries is easy. All you have to do is open the source file in the editor, and double click the mouse in the left side bar, called *marker* bar, of the statements you would like to step into. You can remove breakpoints by double-clicking the breakpoint icon.

Figure 8-72 shows that an icon is placed next to the line of code where a breakpoint is set.

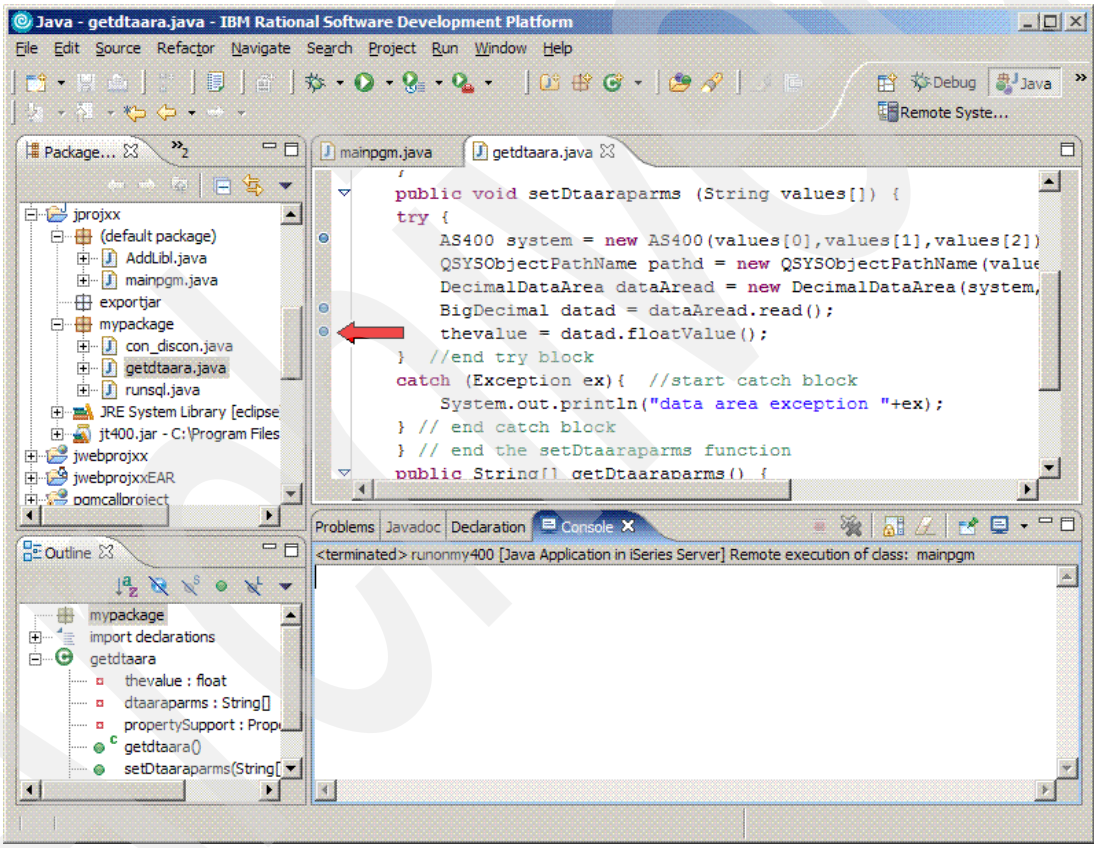


Figure 8-72 Setting breakpoints in WebSphere Development Studio Client for iSeries

Test the application locally

Before testing the application on the System i platform, it makes sense to ensure that the application runs in the local workstation test environment. This is also very easy in WebSphere Development Studio Client for iSeries. All you have to do is right-click the class (which must have a main() method) and select **Run** → **Java Application**, as shown in Figure 8-73.

See Figure 8-73.

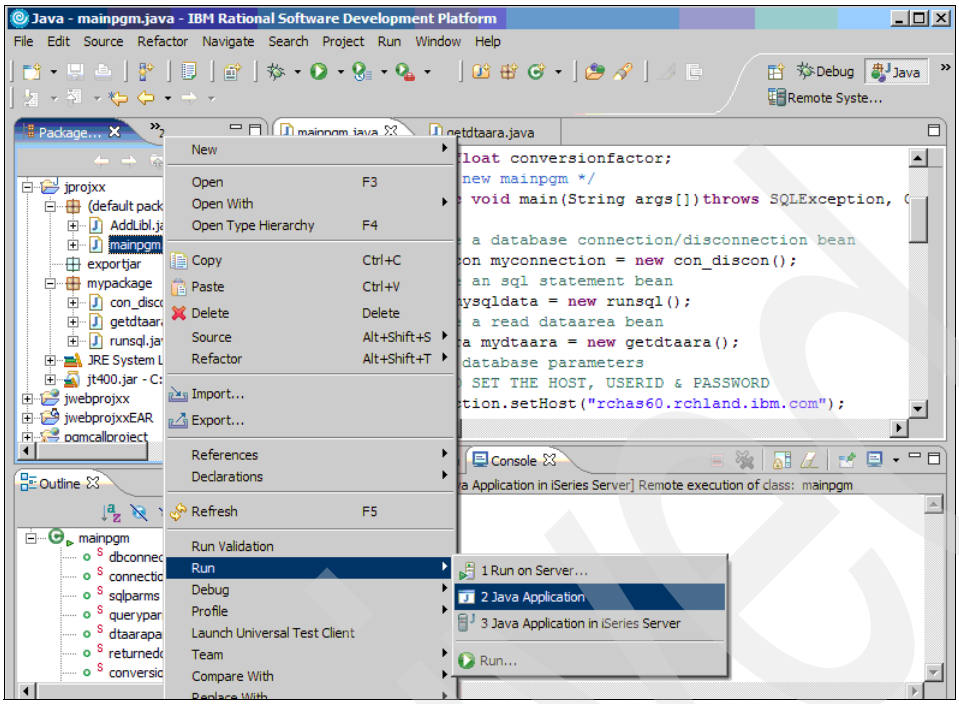


Figure 8-73 Testing an application using the workstation JVM

If there is no graphical user interface, all input and output is done at the console. If there are graphics they must open in a window on your workstation. You can check the console for any error messages or output data as shown in Figure 8-74.

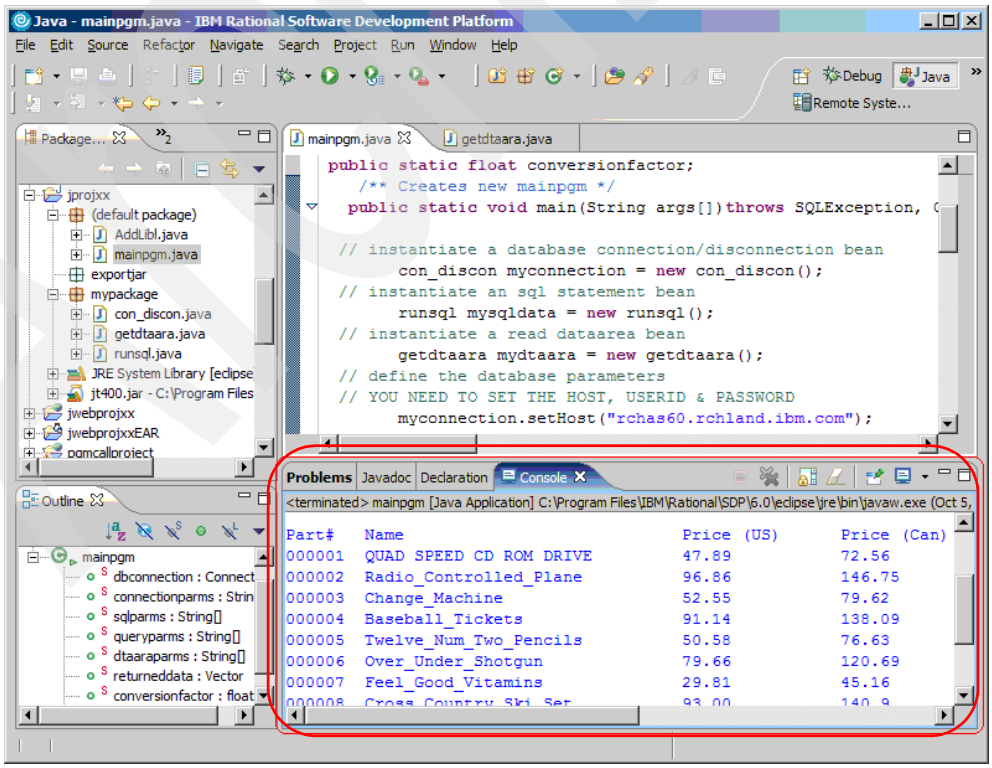


Figure 8-74 Application outputs open in the console

Define a remote system explorer connection to the System i platform

In the previous step you tested the application in the local workstation JVM. WebSphere Development Studio Client for iSeries allows you to debug the application directly on System i platform. You have to define a remote system connection to the System i platform if you want to do this. The remote system connection provides the communications link between the workstation development environment and the System i platform.

If you have already created a remote system explorer (RSE) connection to your System i platform you can ignore this section. Otherwise, these are the steps you have to perform:

1. At your WebSphere Development Studio Client for iSeries workspace open the remote system explorer perspective by selecting **Window** → **Open Perspective** → **Other...** → **Remote System Explorer**.
2. At the top of the Remote Systems view expand **New Connection**.
3. Right-click **iSeries...** and select **New connection**.

   If you have an existing profile it must open as the parent profile option. You can use that, or specify a new profile, as shown in Figure 8-75.

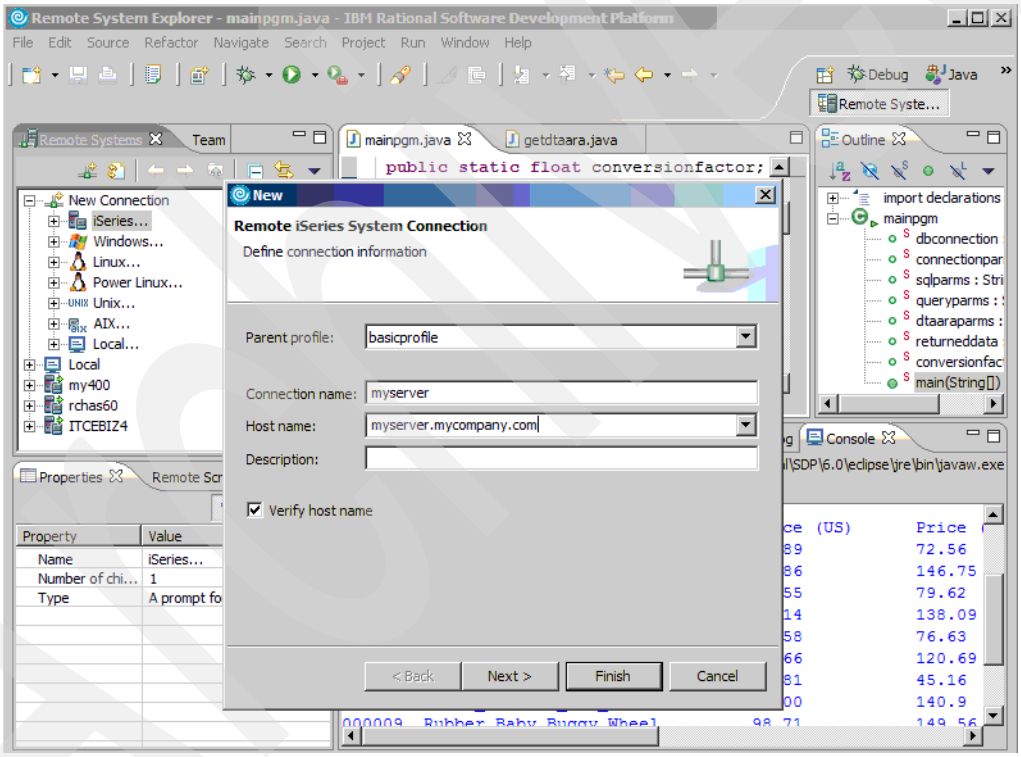


Figure 8-75 Define a remote connection to System i platform

4. Enter a descriptive value for the location name, and the host name or TCP/IP address of your System i platform.
5. Click **Finish**.
6. In the Remote Systems view right-click your server connection and select the **Connect** option. Enter an appropriate user profile and password. You must be connected. You can view library objects, the System i file systems, and other capabilities.

See Figure 8-76.

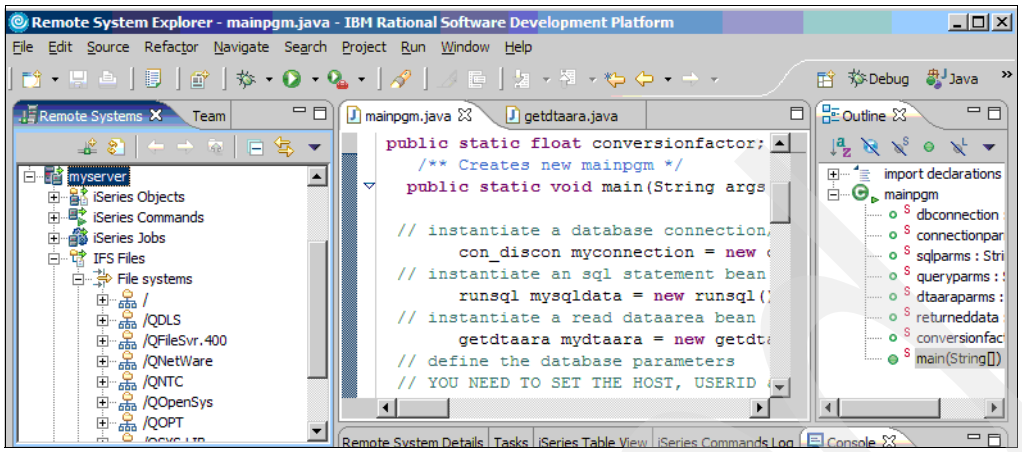


Figure 8-76 Connected to System i platform

7. Now that you have connected to the server you can copy the application to the System i platform.

Copy your application to the System i file system

Perform the following steps:

1. Return to the Java perspective by clicking on the **Java perspective** icon, as shown in Figure 8-77.

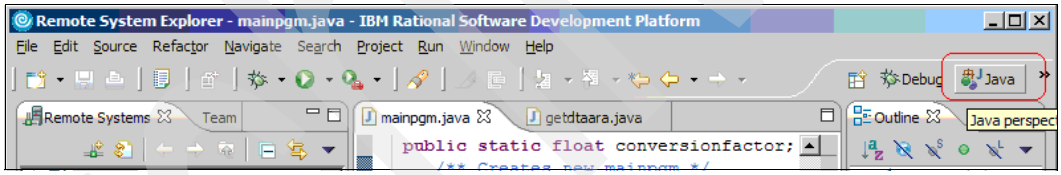


Figure 8-77 Switching between perspectives in WDSc for i5/OS

2. Right-click your Java application project and select the **Export** option.
3. You must see numerous options, including file system (mapped network drive), FTP, and remote file system (remote system explorer connection created earlier). To use the connection created earlier, select the **Remote file system** option and click **Next**.
4. You must see all the objects in your local project selected (check boxes selected). Click the **Browse...** button next to the *To directory* field.

5. Expand the link to the appropriate server and navigate to a directory on the System i platform. In this example the project is copied to the /home directory, as shown in Figure 8-78.

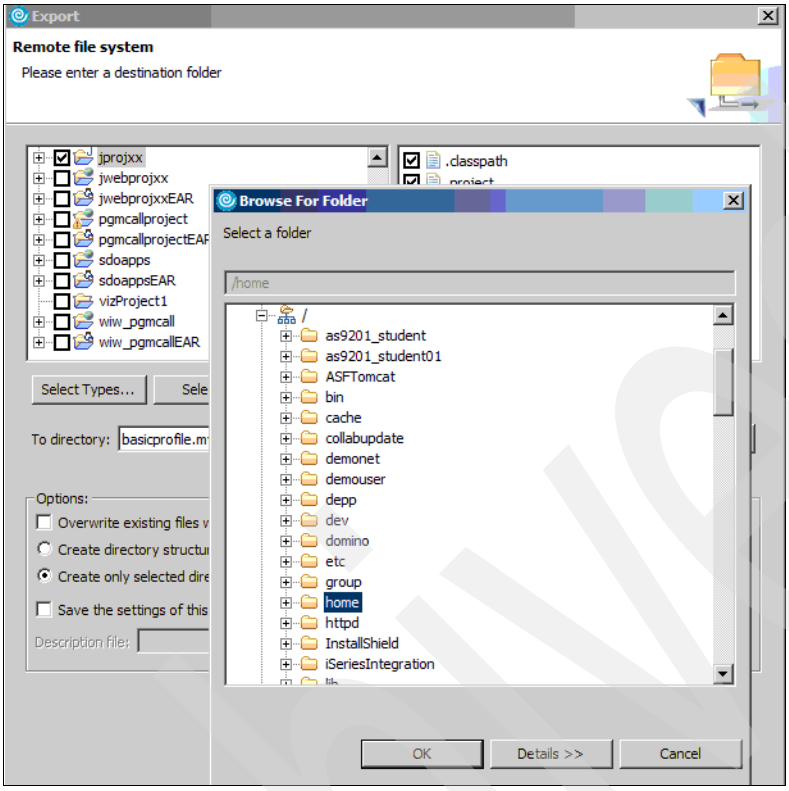


Figure 8-78 Specify a location for the Java program objects

6. Click the **OK** button, then **Finish** to copy the files to the System i file system.
7. You can use a mapped network drive or the i5/OS WRKLNK command to verify that the Java classes and other resources have been copied to the System i file system.

   Now that the objects are on the System i file system you can run the application in debug mode.

Start a debug session on the System i platform

Perform the following steps:

1. At your WebSphere Development Studio Client for iSeries session locate your Java project and select the application with the appropriate main() method.
2. At the main menu select **Run** → **Debug...** option.
3. At the Debug window select the **Java Application in iSeries Server** option. Click the **New** button.
4. Specify an appropriate name for the debugging session.
5. Click the **Browse...** button next to the *Remote iSeries working folder* field.

See Figure 8-79.

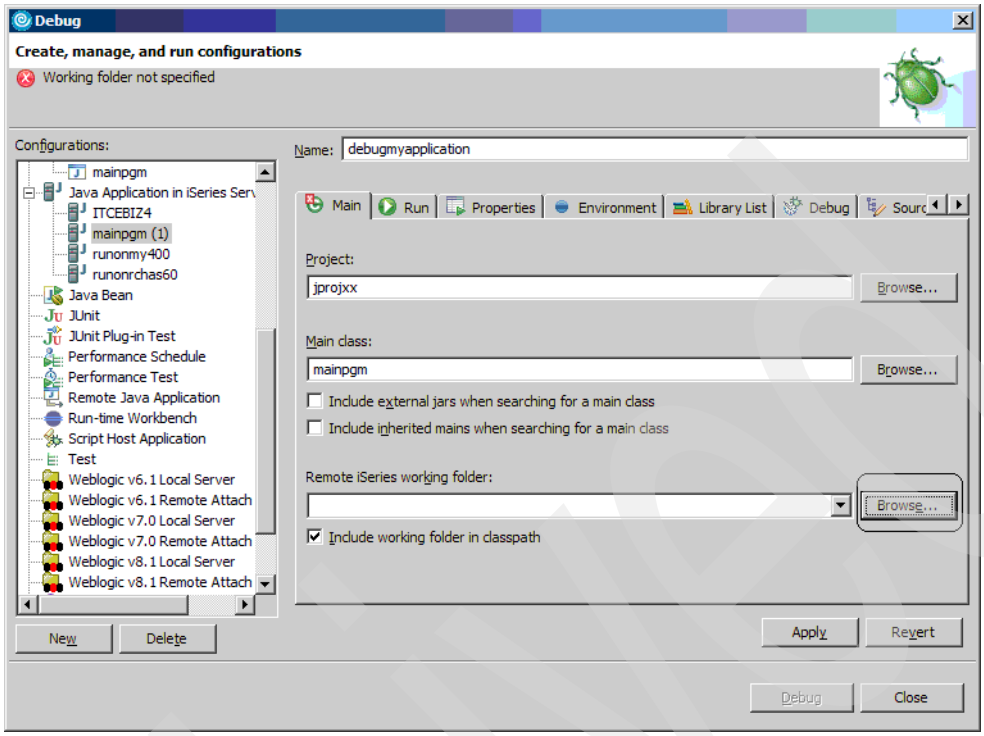


Figure 8-79 Launch the application in debug mode on System i platform

6. At the *Browse for folder* window expand the remote system connection link you specified earlier and locate the folder you copied the Java application to and click **OK**.

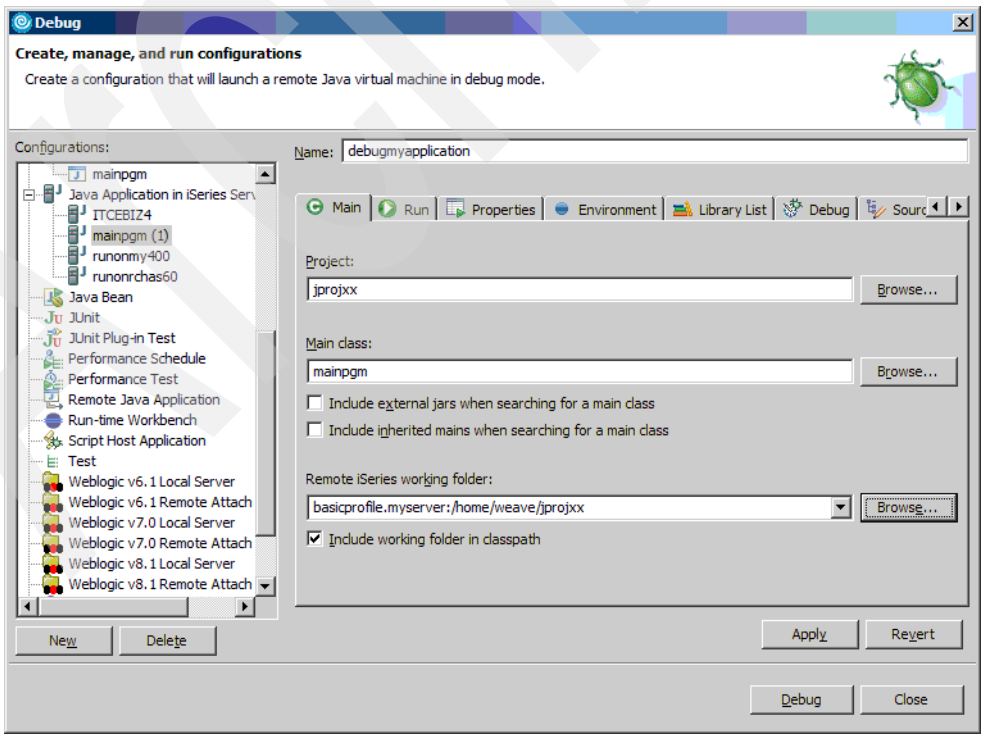


Figure 8-80 Select the folder containing the Java application

Note: The working folder's format is as follows:

`RSEProfile.ConnectionName:directory`

See Figure 8-80. Note that the **Include working folder in classpath** option is selected.

7. Click the **Properties** tab. Enter `java.version` for the property name and `1.5` for the value.
8. Click the **Append** button. This ensures that JDK 1.5 is used on System i platform. See Figure 8-81.

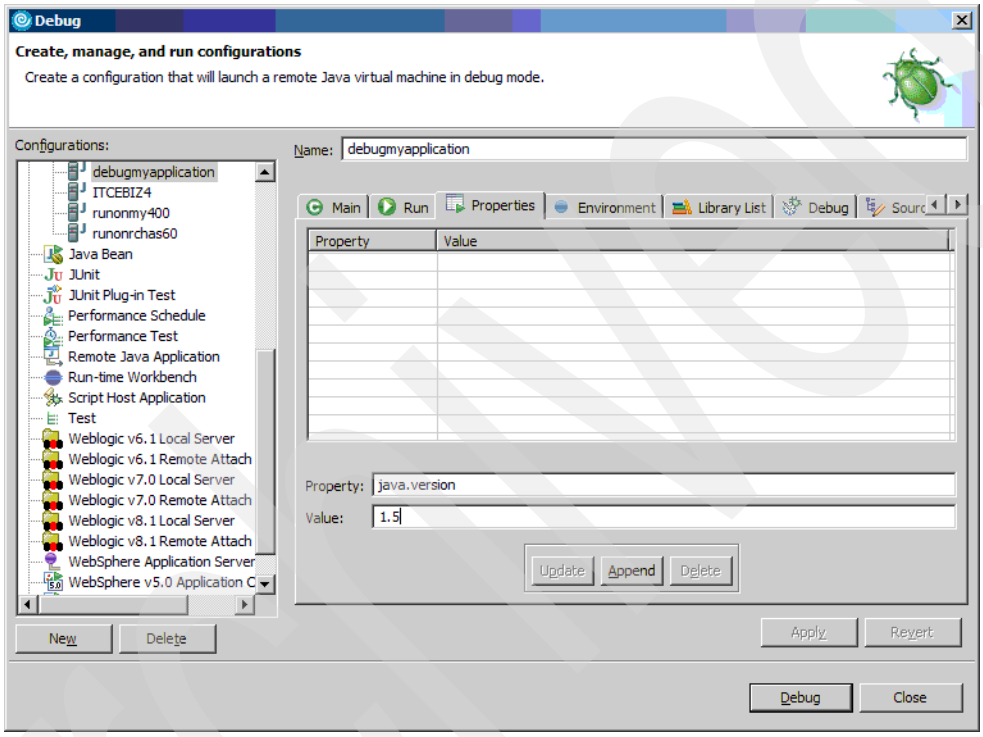


Figure 8-81 Ensure that JDK 1.5 is used on System

Restriction: At the time of this writing, only Classic JVM can be used for remote debugging on System i platform.

9. The classpath includes the working folders you uploaded to the System i platform. If you have to specify additional classpath information or other environment variables, click the **Environment** tab and specify each variable and the value. In this example the IBM Toolbox for Java has been added.

See Figure 8-82.

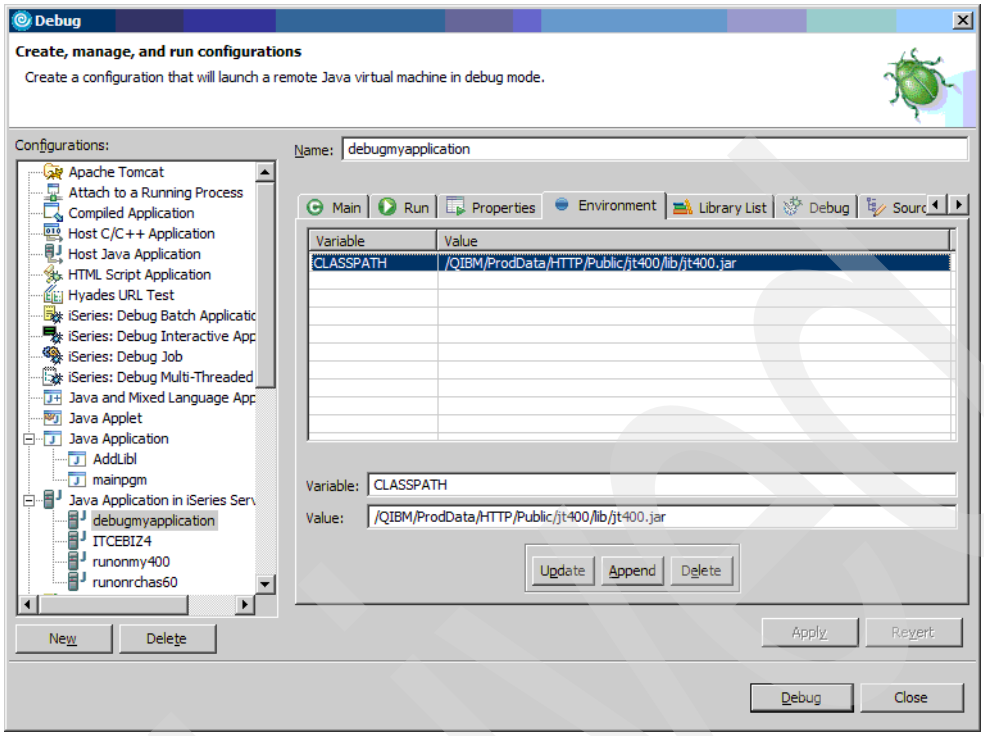


Figure 8-82 Set the CLASSPATH environment variable

10. Enter the appropriate properties, environment variables, library list, and any other necessary parameters. Then click **Apply**.
11. Click the **Debug** button. You might be prompted to enter a user profile and password to connect to the System i platform. Enter the necessary values and click **OK**.
12. You must see a progress window open, then another window asking you to switch to the debug perspective. Click **Yes**.

Step through the application

Perform the following steps:

1. You must now be in the Debug perspective and see your application paused at the first breakpoint you defined.
2. In the debug view you must see the application thread in a suspended state, awaiting you to step through the application. In the console view you must see the i5/OS job information as shown in Figure 8-83.

See Figure 8-83.

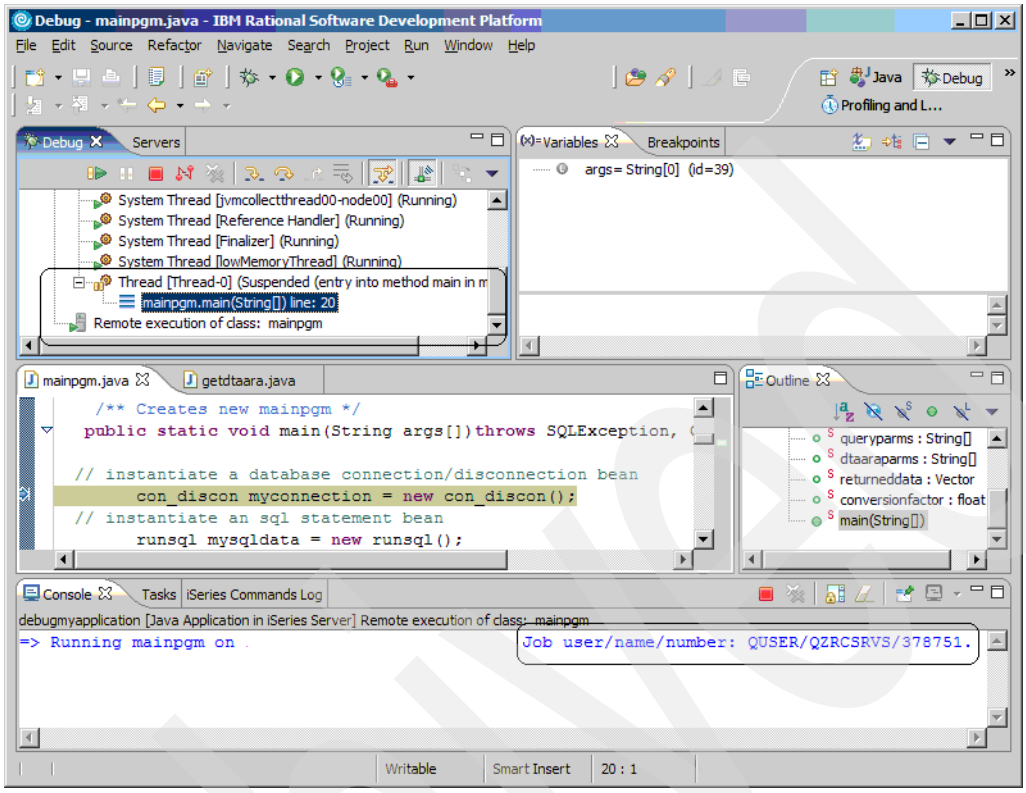


Figure 8-83 Debug session for a Java application running on System i platform

3. If you are interested in looking at the i5/OS job details, start a 5250 emulator session and sign in. You can then use the WRKACTJOB or WRKJOB command to view the job log.

   Figure 8-84 shows an example of what you must see.

```
                              Display Job Log
                                                          System:   MYSERVER
Job . . :   QZRCSRVS      User . . :   QUSER         Number . . . :   378751

     Job 378751/QUSER/QZRCSRVS started on 10/06/06 at 10:53:32 in subsystem
       QUSRWRK in QSYS. Job entered system on 10/06/06 at 10:53:32.
     Printer device PRT01 not found.
     ACGDTA for 378751/QUSER/QZRCSRVS not journaled; reason 1.
     Job changed successfully; however errors occurred.
     User USERID from client 10.10.136.186 connected to server.
     Printer device PRT01 not found.
     ACGDTA for 378751/QUSER/QZRCSRVS not journaled; reason 1.
     Job changed successfully; however errors occurred.
     Library QSYS2924 not found.
     Client request - run program QSYS/QWCRTVCA.
     Client request - run program QSYS/QUSRJOBI.
     Client request - run program QSYS/QWTCHGJB.
     Client request - run program QSYS/QWCRTVCA.
     Client request - run command 'JAVA                          '.
```

Figure 8-84 i5/OS job log for the debugging process

The QZRCSRVS job is not the Java job, but just a job that provides a connection to the debugger.

4. Click the **Resume** icon to advance to the next breakpoint you defined in the source code. See Figure 8-85.

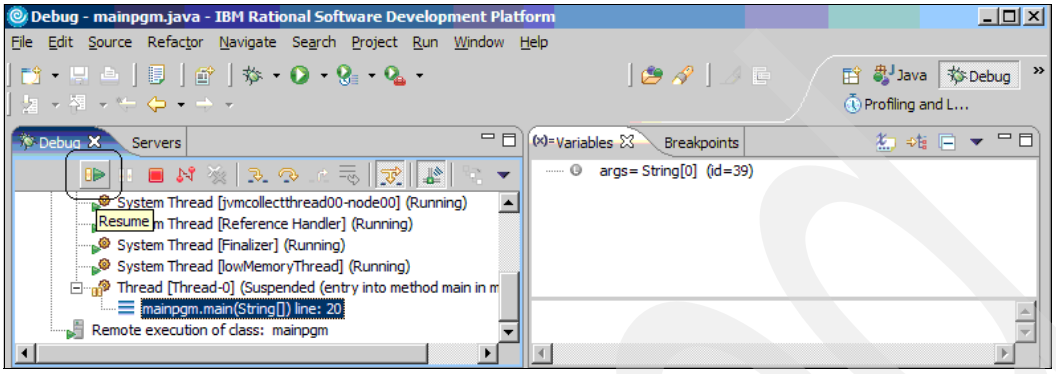


Figure 8-85 Use the Resume action to step through the application

5. Click the **Resume** icon to continue stepping through the application and determine if and where there is an application error. You can also monitor the values of variables and determine the status of each thread. See Figure 8-86.

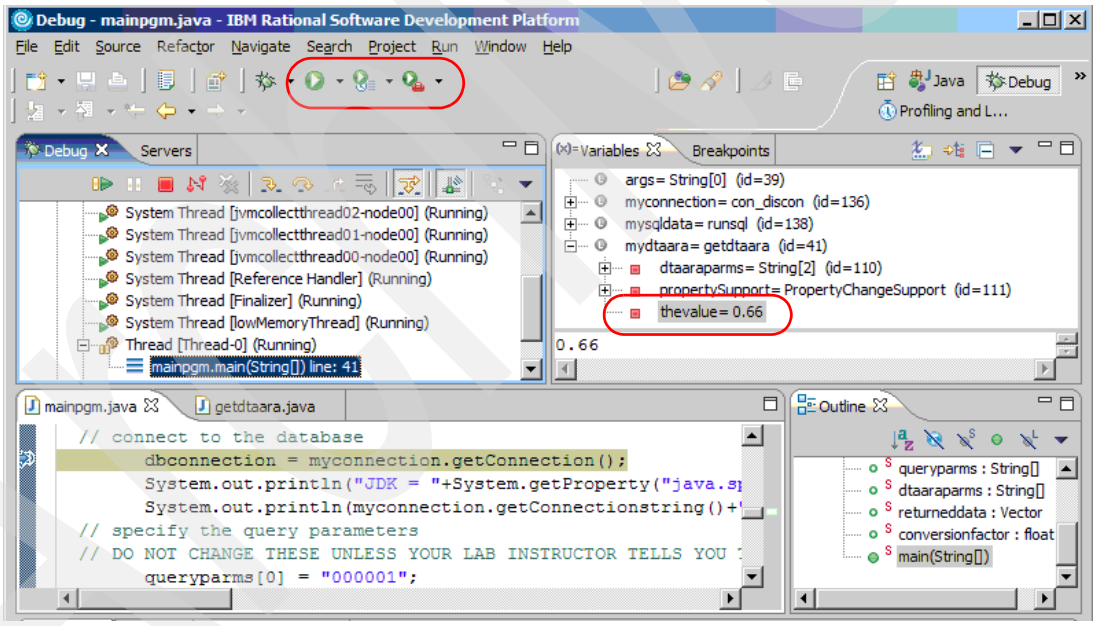


Figure 8-86 View application variables and thread status

6. Figure 8-86 shows an example of the mainpgm class instantiating a JavaBean (the getdtaara class). The getdtaara class includes a method that reads a decimal data area on the System i platform, into a variable called “thevalue”. In this example, “thevalue” is .66.

7. You can continue clicking the **Resume** icon to continue stepping through the application (or other icons, like *Step over* or *Step into)* and determine if and where there is an application error. You can also monitor the console output.

See Figure 8-87.

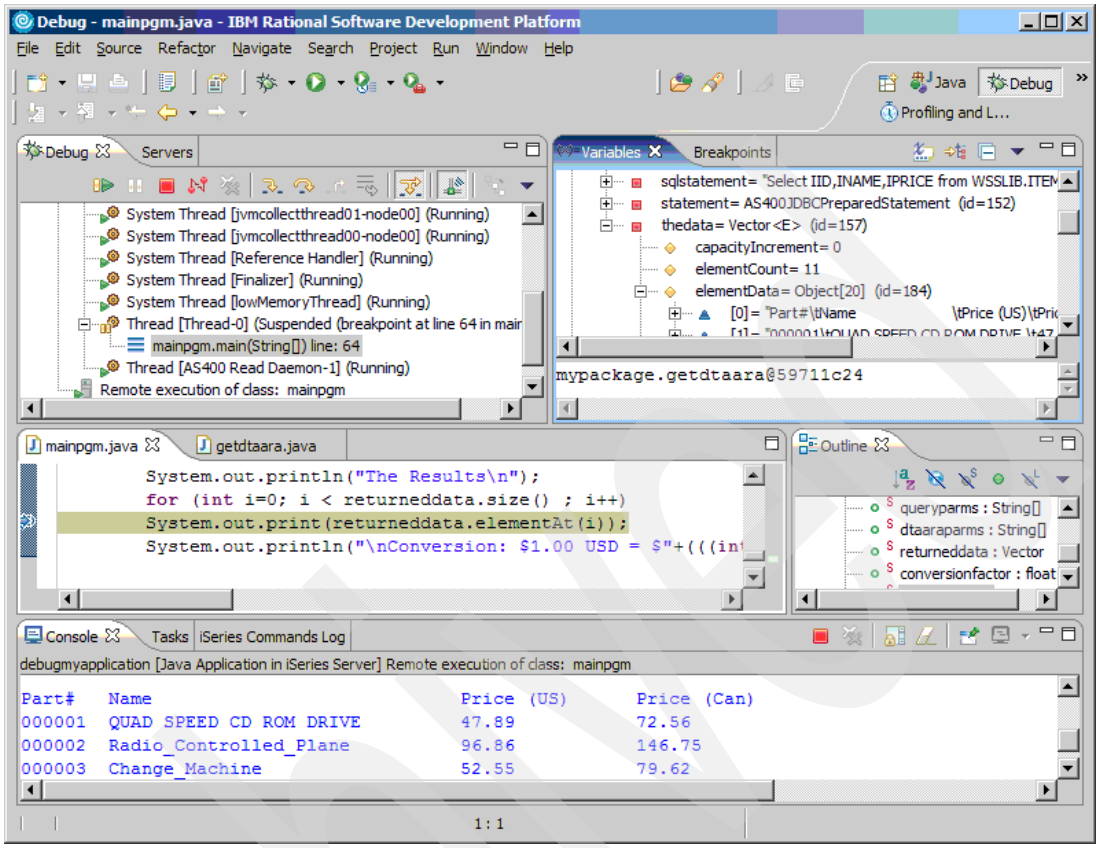


Figure 8-87 View console output and application variables

8. Depending on how many and where the breakpoints were set, you might have to experiment some to determine the exact cause of the application problem.

   You can also use the remote debugging function to get an idea of the application response time components.

A

Running WebSphere Application Server with IBM Technology for JVM

This appendix describes the usage of the IBM Technology for JVM with WebSphere Application Server on System i platform. The topics covered are:

System requirements for i5/OS

WebSphere Application Server version 6.1 is supported on both Classic JVM and IBM Technology for JVM on i5/OS V5R4M0 or later releases. Earlier versions of WebSphere Application Server are only supported on Classic JVM. i5/OS releases prior to V5R4M0 do not support IBM Technology for JVM. These are the basic requirements for WebSphere Application Server version 6.1 running on V5R4M0. For a complete list of hardware and software prerequisites refer to:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.base.iseries.doc/info/iseries/ae/cins_is_prqsvr.html

Some of the prerequisites are listed as follows:

- i5/OS V5R4M0 and cumulative program temporary fix (PTF) package C6192540 or later
- Java Developers Kit (JDK) V5.0
  - 5722-JV1 option 7
  - 5722-JV1 option 8
- i5/OS Qshell utilities (5722-SS1 option 30)
- i5/OS Host Servers (5722-SS1 option 12)
- i5/OS Extended Base Directory Support (5722-SS1 option 3)
- i5/OS Portable Application Solutions Environment (5722-SS1 option 33)
- WebSphere Application Server version 6.1 group PTF SF99323 version 5 or later
  - DB2 for i5/OS group PTF SF99504 version 4 or later
  - Java group PTF SF99291 version 3 or later
  - HTTP server group PTF SF99114 version 3 or later

Important: If you install any of the listed products after installing the i5/OS cumulative PTF package, you must reinstall the cumulative PTF package to ensure all required PTF are applied. Reinstall the WebSphere Application Server group PTF if you installed 5722-JV1 option 7, 8, or 5722-SS1 option 33 (i5/OS PASE) after installing group PTF.

Refer to the WebSphere Application Server group PTF cover letter for installation instructions. Ensure you run the *update* script, which actually applies the WebSphere Application Server fixes.

Switching between JVMs in WebSphere Application Server 6.1

WebSphere Application Server 6.1 for i5/OS can run with Classic JVM and IBM Technology for JVM. It includes a Qshell script than can switch between JVMs. This section covers:

- The *enableJVM* script that specifies the JVM for one or more WebSphere Application Server profiles
- An example of stopping a profile, changing the JVM and restarting the profile
- How to determine which JVM is in use

The enableJVM script

By default, WebSphere Application Server version 6.1 specifies Classic JVM for all profiles that are created.

Attention: WebSphere Application Server 6.1 supports installing the product multiple times on the same server or logical partition. To get a list of all of your installed environments including the default profile location and install library, you can use the following script:

```
/QIBM/ProdData/WebSphere/AppServer/V61/base/bin/querywasinstalls
```

In these examples the default installation location is assumed.

WebSphere Application Server version 6.1 for iSeries includes an i5/OS Qshell script that can switch the JVM type, if you do not use the default installation directory, adjust these instructions accordingly:

```
/QIBM/ProdData/WebSphere/AppServer/V61/Base/bin/enableJvm
```

This script requires the user to have *ALLOBJ authority. The command syntax is:

```
enableJVM parameters
```

Here `parameters` are:

- -jvm [std32 | classic]

  This is a required parameter that specifies the JVM in use:

  - std32 = IBM Technology for JVM
  - classic = Classic JVM

- -profile <profile_to_be_enabled>

  This is an optional parameter that specifies the profile to enable.

- -verbose

  This is an optional parameter that enables additional trace statements.

Important: If a profile is not specified, *all* profiles are switched to the specified JVM.

Here are several examples:

- `enableJVM -jvm std32 -profile default`

  This sets the default profile to run on IBM Technology for JVM.

- `enableJVM -jvm classic`

  This sets ALL profiles to run on Classic JVM.

- `enableJVM -jvm std32 -profile default -verbose`

  This sets the default profile to run on IBM Technology for JVM and display additional details on the screen.

Attention: With WebSphere Application Server Network Deployment V6.1 it is possible to include servers with Classic JVM and IBM Technology for JVM in the same cell.

Enable an existing WebSphere Application Server profile with IBM Technology for JVM

These instructions provide a complete set of steps for switching to IBM Technology for JVM:

1. Sign in to the System i platform with an appropriate userID and password.
2. Enter STRQSH on the CL command line and hit Enter.
3. At the Qshell prompt, run the following command:

   ```
   cd /QIBM/ProdData/WebSphere/AppServer/V61/Base/bin
   ```

4. If your profile is running, stop it (we use the default profile) as shown in Figure A-1:

   ```
   stopServer -profileName default
   ```

```
                                QSH Command Entry

  $
 > cd /QIBM/ProdData/WebSphere/AppServer/V61/Base/bin
  $
 > stopServer -profileName default
  ADMU0116I: Tool information is being logged in file

/QIBM/UserData/WebSphere/AppServer/V61/Base/profiles/default/logs/server1/stopS
erver.log
  ADMU0128I: Starting tool with the default profile
  ADMU3100I: Reading configuration for server: server1
  ADMU3201I: Server stop request issued. Waiting for stop status.
  ADMU4000I: Server server1 stop completed.
===>
F3=Exit  F6=Print F9=Retrieve F12=Disconnect
F13=Clear  F17=Top  F18=Bottom  F21=CL command entry
```

Figure A-1 Ending the profile

5. Invoke the enableJVM script to change your profile to use the IBM Technology for JVM:

   ```
   enableJVM -profile default -jvm std32
   ```

   See Figure A-2.

```
                                QSH Command Entry

> enableJVM -profile default -jvm std32
  ADEJ0014I: Enabling profile default to use the specified JVM.
  ADEJ0002I: Success: The profile will now use the specified JVM.
  $
===>
F3=Exit  F6=Print F9=Retrieve F12=Disconnect
F13=Clear  F17=Top  F18=Bottom  F21=CL command entry
```

Figure A-2 Enable JVM for a specific server profile

6. Start the WebSphere Application Server profile that you changed earlier.

   ```
   startServer -profileName default
   ```

   See Figure A-3.

```
                                QSH Command Entry

> startServer -profileName default
 CPC1221: Job 089988/QEJBSVR/SERVER1 submitted to job queue QWASJOBQ in
  library QWAS61.
   CWNATV00I: Application server server1 in profile default has started and is
ready to accept connections on admin port 9060.
===>
F3=Exit  F6=Print F9=Retrieve F12=Disconnect
F13=Clear F17=Top  F18=Bottom  F21=CL command entry
```

Figure A-3 Starting the profile

7. Your server must now run in IBM Technology for JVM.

Determining the JVM in use

There are two basic ways to determine which JVM is used by a WebSphere Application Server profile:

- View the job log for your WebSphere Application Server profile.
- View the SystemOut.log file for your WebSphere Application Server profile.

The job log option is the easier one to use and is illustrated.

View the job log

To view the job log, perform the following instructions:

1. Find your active profile by using the WRKACTJOB SBS(QWAS61) command.
2. Use option 5, work with the job, then option 10, display job log, for the application server job.
3. Press the F10 key to display the detailed messages. If you use Classic JVM, you see the message shown in Figure A-4.

See Figure A-4.

```
                              Display Job Log
                                                              System:   RCHAS60
 Job . . :   MYAPPSVR02    User . . :   QEJBSVR       Number . . . :   113772

     Job 113772/QEJBSVR/MYAPPSVR02 started on 08/11/06 at 15:14:44 in subsystem
       QWAS61 in QWAS61. Job entered system on 08/11/06 at 15:14:44.
     ACGDTA for 113772/QEJBSVR/MYAPPSVR02 not journaled; reason 1.
     Java Virtual Machine is Classic.
     Server starting with user profile QEJBSVR and JDK 1.5.0.03-_13_feb_2006.
     Statistics elapsed time reset.
     WebSphere application server MyAppSvr02 ready.

Bottom
 Press Enter to continue.

 F3=Exit   F5=Refresh   F10=Display detailed messages   F12=Cancel
 F16=Job menu           F24=More keys
```

Figure A-4 Display messages for Classic JVM

If it is IBM Technology for JVM, then you see the message shown in Figure A-5.

```
                              Display Job Log
                                                              System:   RCHAS60
 Job . . :   MYAPPSVR02    User . . :   QEJBSVR       Number . . . :   113772

     Job 113772/QEJBSVR/MYAPPSVR02 started on 08/11/06 at 15:14:44 in subsystem
       QWAS61 in QWAS61. Job entered system on 08/11/06 at 15:14:44.
     ACGDTA for 113772/QEJBSVR/MYAPPSVR02 not journaled; reason 1.

     Java Virtual Machine is IBM Technology for Java. PID(343)
     Server starting with user profile QEJBSVR and JDK 1.5.0.

     Statistics elapsed time reset.
     WebSphere application server MyAppSvr02 ready.

Bottom
 Press Enter to continue.

 F3=Exit   F5=Refresh   F10=Display detailed messages   F12=Cancel
 F16=Job menu           F24=More keys
```

Figure A-5 Display messages for IBM Technology for JVM

4. Note that the process identifier PID(343) is shown in the job log. You can use this information in case you have to do a heap dump or other diagnostics described in Chapter 8, “Analyzing JVM behavior” on page 83.

Configuring the WebSphere Application Server JVM

Configuring the IBM Technology for JVM that is used with WebSphere Application Server is easy if you use the administrative console:

1. At a console session click **Servers** → **Application Servers** → ***<application server name>***.
2. At the application servers configuration panel click **Java and Process Management** → **Process Definition** → **Java Virtual Machine**. See Figure A-6.

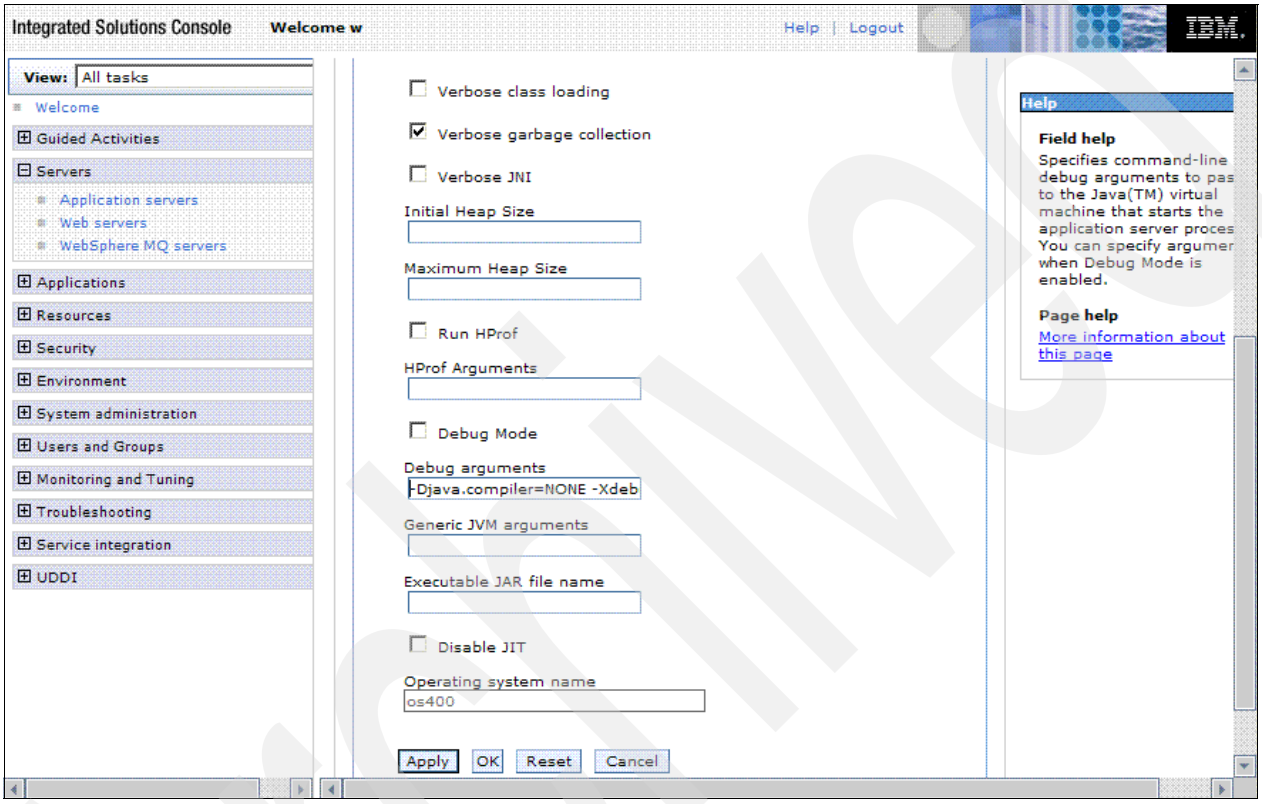


Figure A-6 WebSphere Application Server V6.1 console makes it easy to customize JVM properties

3. In this example you can see that verbose garbage collection has been specified. If you see no values in the Initial Heap Size and Maximum Heap Size fields, then the default values are used:

   ```
   initial heap size: 50 MB
   maximum heap size: 256 MB
   ```

   These are defined in the following file:

   ```
   <WAS_install_root>/<edition>/classes/properties/os400j9.systemlaunch.properties
   ```

4. You can change these if you specify the appropriate parameters and click the **Apply** button. You are then prompted to save or review the changes. Click the **Save** link after ensuring all parameters have been entered correctly.
5. Note that the generic JVM arguments parameter is blank in Figure A-6. This tells you that the default garbage collection policy (optimum throughput) is in place.

These are some commonly used JVM arguments for the WebSphere Application Server version 6.1 environment:

- -agentlib:QWASJVMTI (enables JVM profiling)
- -Xgcpolicy:optthruput (specifies the optimum throughput garbage collection policy)
- -Xgcpolicy:optavgpause (specifies the minimum average pause garbage collection policy)
- -Xgcpolicy:gencon (specifies the generational garbage collection policy)
- -Xgcpolicy:subpool (specifies the subpool garbage collection policy)

Because IBM Technology for JVM uses the just-in-time (JIT) compiler exclusively, you do not specify the -Djava.compiler=jitc parameter like you would with Classic JVM.

Attention: Be very careful when you specify JVM command line arguments. Mistakes as simple as case sensitivity in the parameters can prevent the JVM from starting. JVM command line arguments are specified in the profile's server.xml file.

JVM parameters exclusive to Classic JVM may prevent the IBM Technology for JVM from starting, and vice versa.

Important: At the time of this writing, the JVM profiling functions were not working.

6. Restart your WebSphere profile to enable new parameters.

Monitoring the WebSphere Application Server JVM

The measurement and analysis tools discussed in Chapter 8, "Analyzing JVM behavior" on page 83 can be utilized with a WebSphere Application Server profile that utilizes IBM Technology for JVM.

Monitoring with Tivoli Performance Viewer

This section illustrates how you can use the Tivoli Performance Viewer, which is integrated in the WebSphere Application Server administrative console, to monitor JVM performance. These concepts are applicable to IBM Technology for JVM and Classic JVM:

1. At a console session click **Monitoring and Tuning** → **Performance Monitoring Infrastructure (PMI)** → ***<application server name>***.
2. At the Performance Monitoring Infrastructure (PMI) panel you must be in the configuration tab. This specifies the monitoring settings when the application server starts. The default settings are PMI enabled, with the basic statistic set.

See Figure A-7.

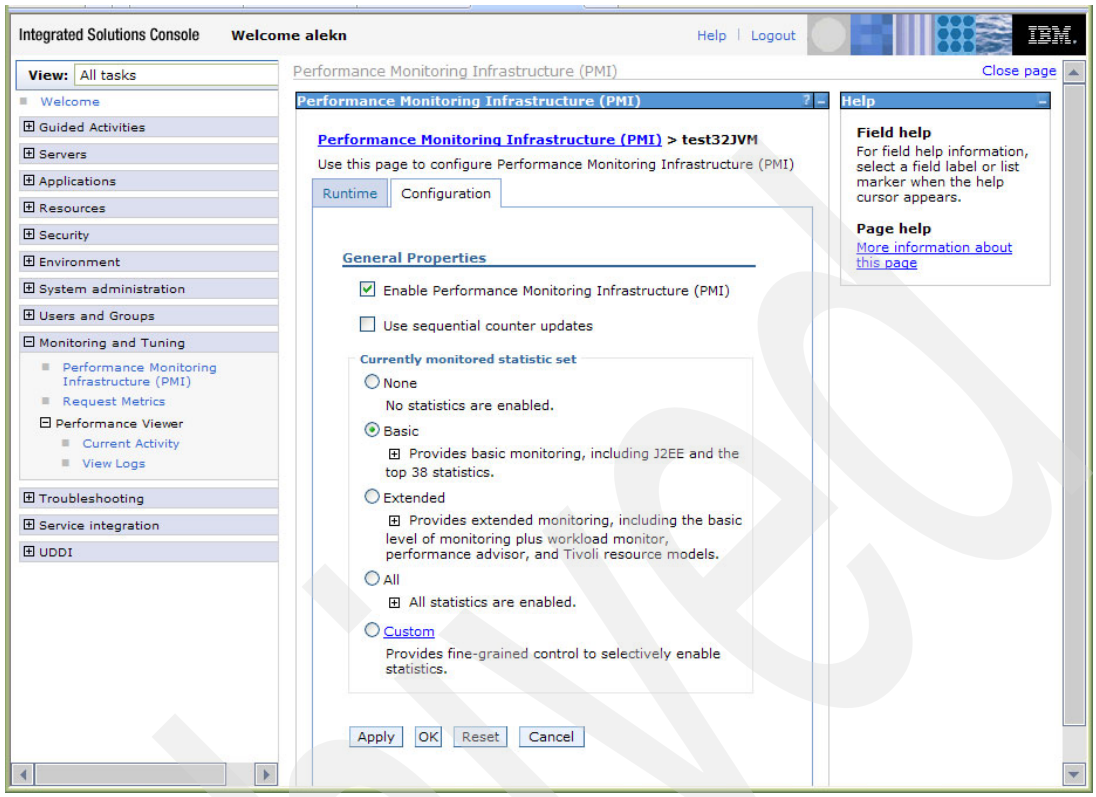


Figure A-7 Configuring PMI

3. Click the **Runtime** tab. This enables you to dynamically specify the performance monitoring policies.
4. Expand the basic monitoring statistic link and scroll down to the JVM options. You must see the following JVM metrics available for monitoring:
   - JVM Runtime.ProcessCpuUsage (the percentage of CPU usage of the JVM runtime)
   - JVM Runtime.UpTime (the total time, in seconds, that the JVM has been running)
   - JVM Runtime.UsedMemory (the used memory, in KB, of the JVM runtime
   - JVM Runtime.HeapSize (the total memory, in KB, of the JVM runtime)

   These metrics are available if JVM profiling is enabled:

   - Garbage Collection.GCTime
   - Garbage Collection.GCIntervalTime
   - Garbage Collection.GCCount
   - Object.ObjectFreedCount
   - Object.ObjectAllocateCount
   - Object.ObjectMovedCount
   - Thread.ThreadEndedCount
   - Thread.ThreadStartedCount
   - Monitor.WaitForLockTime
   - Monitor.WaitsForLockCount
5. At your console session click **Performance Viewer** → **Current Activity** → ***<application server name>***.

6. Expand the **Performance Modules** link and click the **JVM Runtime** box.

   See Figure A-8.

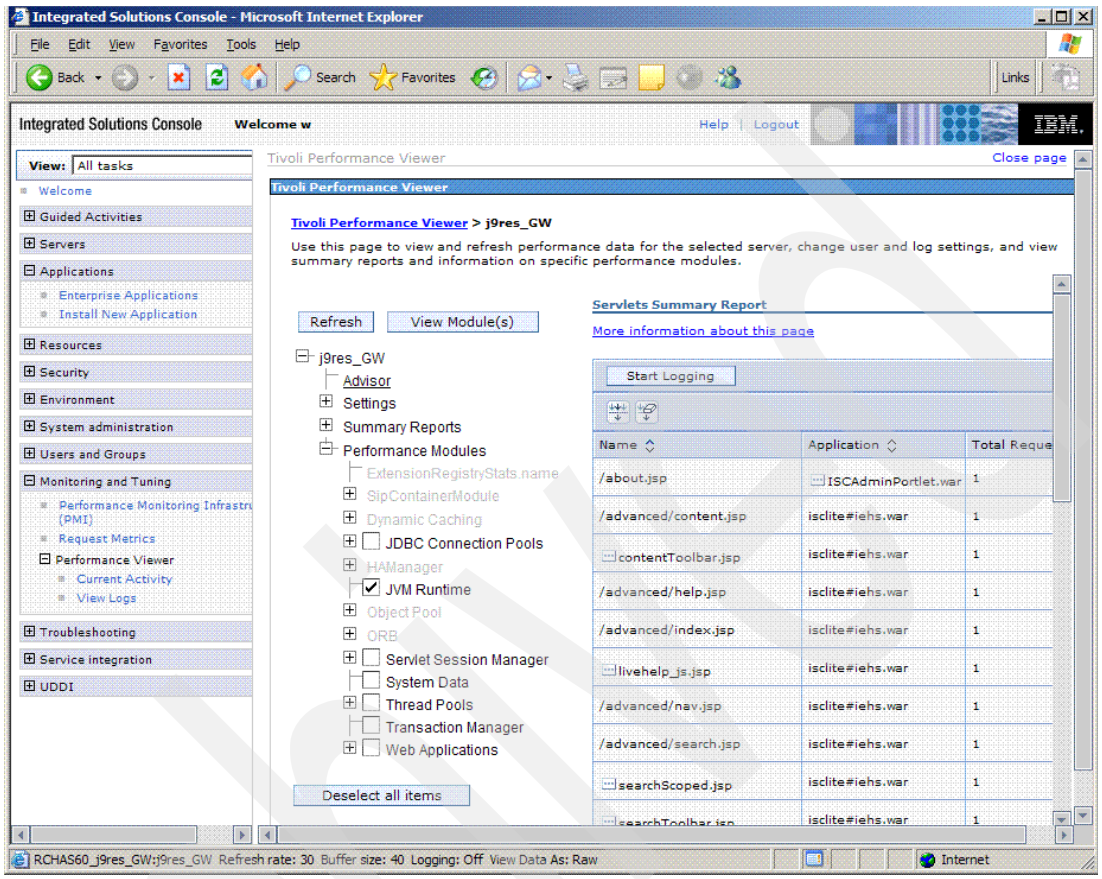


Figure A-8 Select the JVM runtime statistics for viewing

7. Click the **View Module(s)** button to see the JVM runtime data.
8. You must see the JVM statistics in either tabular or graphic format. The following are the requirements for viewing graphics:
   - The Adobe Scalable Vector Graphics plugin for your browser
   - i5/OS PASE (required for IBM Technology for JVM)
   - Set the following parameters:

     ```
     -Djava.awt.headless=true
     -Dos400.awt.native=true
     ```

   - In the following file:

     ```
     os400j9.systemlaunch.properties
     ```

     This file is located in:

     ```
     <WAS_install_root>/<edition>/classes/properties/os400j9.systemlaunch.properties
     ```

See Figure A-9.

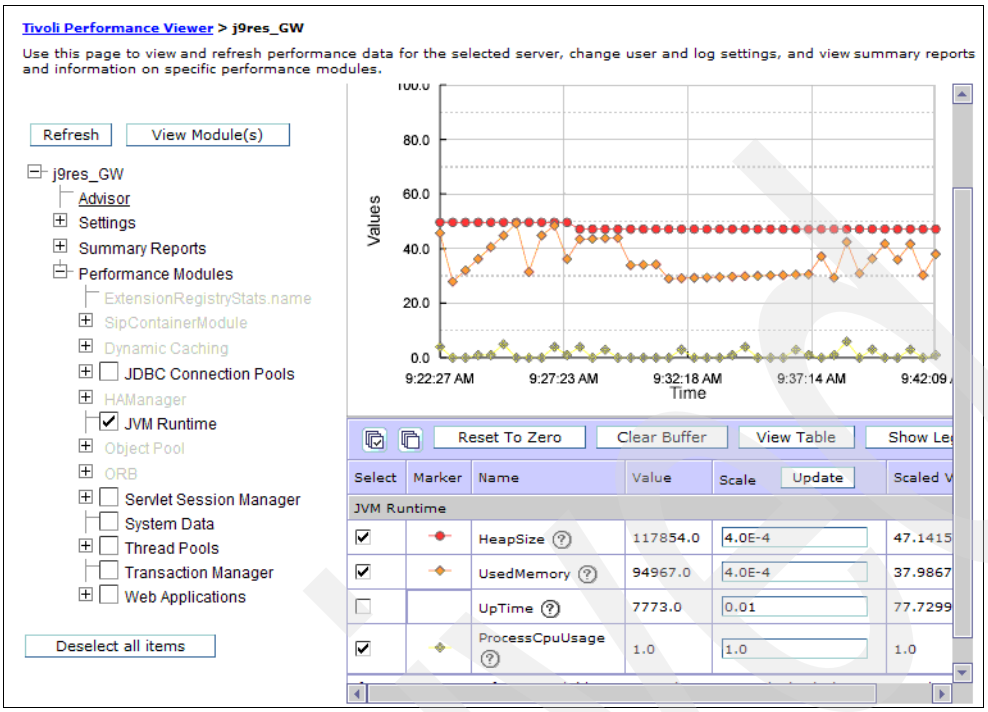


Figure A-9 View a graphical representation of the JVM runtime

9. In Figure A-9 you can also see that the total heap size is relatively constant and that CPU usage is minimal.
10. Figure A-9 shows an example of the JVM runtime data in graphical format. In this case the graph scaling was modified to improve the view in order to see how the used memory changes.
11. Figure A-10 shows an example of the JVM runtime data in tabular format.

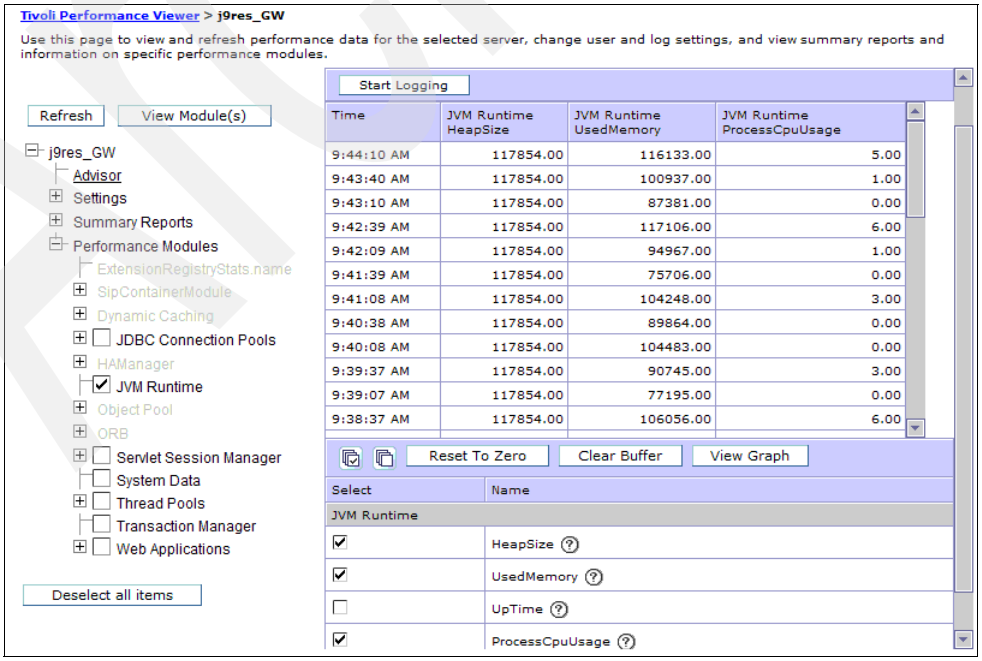


Figure A-10 View a tabular representation of the JVM runtime

12. You can use the Tivoli Performance Viewer to get a conceptual view of your JVM's memory usage. If the memory usage is fairly consistent over time, it is a good indicator that currently there are no memory leaks in your applications.

    On the other hand, if memory usage continues to grow, even with a steady workload, it might indicate a memory leak and you have to utilize the techniques in Chapter 8, "Analyzing JVM behavior" on page 83 to analyze the problem in more detail.

Verbose garbage collection and logging

You can enable verbose garbage collection for IBM Technology for JVM or Classic JVM using the WebSphere Application Server administrative console and selecting the **Verbose garbage collection** check box as indicated in Figure A-6 on page 173. By default verbose garbage collection statistics is written to the following directory:

```
<WAS_install_root>/<edition>/profiles/<profile>/logs/<profile>
```

In this directory the verbose garbage collection statistics is written to the following files:

- For IBM Technology for JVM: native_stderr.log
- For Classic JVM: native_stdout.log

The SystemOut.log file in the aforementioned directory includes helpful JVM information. The excerpt in Example A-1 shows that the IBM Technology for JVM is used.

Example: A-1 Excerpt showing the use of IBM Technology for JVM

```
WebSphere Platform 6.1 [BASE 6.1.0.1 cf10631.18]  running with process name
RCHAS60_j9res_GW\RCHAS60_j9res_GW\j9res_GW and process id 212236/QEJBSVR/J9RES_GW
Host Operating System is OS/400, version V5R4M0
Java version = J2RE 1.5.0 IBM J9 2.3 OS400 ppc-32  (JIT enabled)
J9VM - 20060501_06428_bHdSMR
JIT  - 20060428_1800_r8
GC   - 20060501_AA, Java Compiler = j9jit23, Java VM name = IBM J9 VM
```

The excerpt in Example A-2 shows the heap monitor status during the server startup, in addition to the i5/OS memory pool details.

Example: A-2 Excerpt showing heap monitor status during server startup

```
Heap Monitor started for 212236/QEJBSVR/J9RES_GW in subsystem QWAS61
in Pool *BASE pool ID=2
Poolsize(MB)=25107 Reserved(MB)=2
Heap total(MB)=50 Free(MB)=19 UsedHeap(MB)=30
MaxHeap(MB)=256
InitHeap(MB)=50
```

Example A-3 is an example of the verbose garbage collection data.

Example: A-3 Verbose garbage collection data

```
<af type="tenured" id="34" timestamp="Mon Oct 09 15:14:49 2006"
intervalms="3510.159">
  <minimum requested_bytes="24" />
  <time exclusiveaccessms="0.092" />
  <tenured freebytes="368128" totalbytes="146692096" percent="0" >
    <soa freebytes="0" totalbytes="146323968" percent="0" />
```

```
    <loa freebytes="368128" totalbytes="368128" percent="100" />
  </tenured>
  <gc type="global" id="34" totalid="34" intervalms="3531.284">
    <refs_cleared soft="8" weak="3" phantom="12" />
    <finalization objectsqueued="168" />
    <timesms mark="234.412" sweep="7.486" compact="0.000" total="242.215" />
    <tenured freebytes="49852368" totalbytes="146692096" percent="33" >
      <soa freebytes="49631184" totalbytes="146470912" percent="33" />
      <loa freebytes="221184" totalbytes="221184" percent="100" />
    </tenured>
  </gc>
  <tenured freebytes="49851672" totalbytes="146692096" percent="33" >
    <soa freebytes="49630488" totalbytes="146470912" percent="33" />
    <loa freebytes="221184" totalbytes="221184" percent="100" />
  </tenured>
  <time totalms="247.748" />
</af>
```

Tip: It is recommended that you collect the verbose GC output to a separate file using the -Xverbosegclog parameter in the *Generic JVM arguments* field. See Figure A-6 on page 173. For further information, refer to 5.2.3, "Verbose GC output" on page 44.

Additional information

The WebSphere Application Server information center also has helpful information. It is available at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.base.iseries.doc/info/welcome_base.html

B

IBM Support Assistant

This appendix discusses *IBM Support Assistant* (ISA) version 3.0.1. ISA is a free, local serviceability workbench that helps users resolve questions and problems with IBM software products. A main goal of ISA is to become a single conduit for IBM customers to retrieve and use serviceability tooling and also provide the core foundation of functions required for self-help. Its features include serviceability tools, concurrent searching, quick access to support-related information, and electronic problem management including automated data collection. All of these functions are delivered together in the IBM Support Assistant application to better enable users for self-help problem determination.

The focus of this appendix is in demonstrating the typical tasks for downloading, installing, and using the tool. For more information visit the ISA Web site at:

http://www.ibm.com/software/support/isa

Overview of IBM Support Assistant

IBM Support Assistant is a free software product intended to help IBM customers to be more productive with IBM products by resolving problems faster.

Important: ISA is the strategic delivery mechanism for IBM Java virtual machine (JVM) tools.

You can use ISA when experiencing a problem. ISA offers resources for self-help that can enable you to identify, assess, and overcome questions or problems without having to contact IBM. When it is necessary to contact IBM, ISA offers resources for fast submission of problem reports and immediate, automated collection of diagnostic data that can accelerate problem resolution.

Install IBM Support Assistant

ISA 3.0.1 is available as free download from the following site:

http://www.ibm.com/software/support/isa/

Updates to the ISA application are delivered via the Updater function within ISA (refer to the documentation delivered with the product).

Prerequisites

The following list describes the operating systems that are supported by ISA:

- Microsoft® Windows® XP SP1, 2000, and 2003 server
- Linux RedHat Advanced Server 3, Linux SuSE 9.0
- HP/UX 11
- Solaris 9
- AIX 5.2 and 5.3

Table B-1 describes Web browsers that are supported by ISA.

Table B-1 Supported Web browsers

Platform	Web browser
Windows	MS Internet Explorer® 6.x
Unix	Mozilla 1.7 and later Mozilla Firefox 1.0.7+
Linux	Mozilla 1.7 and later Mozilla Firefox 1.0.7+

ISA, along with the tools described in this IBM Redbook, requires a minimum of 200 MB of free space for installation.

You can use ISA with i5/OS to gather product information, run problem determination tools, and analyze problems, but you must install ISA on a supported workstation that you can connect to your i5/OS. It is also recommended setting up a integrated file system share to easily access i5/OS files from your workstation.

Downloading IBM Support Assistant from Web

There are two major components to downloading ISA from the Web. You must first download the product itself. Then, you must run the Updater program in ISA to download any tools or features for products you are interested in.

The following steps document how to download ISA to the Windows platform. Modify the steps as required for your operating system. Perform the following steps:

1. Open the following URL in your Web browser:

   `https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=isa`

2. Click **Sign in** to sign in to the Web site using your IBM ID and password. If you do not have one, register by clicking the **register now** link.
3. Select the latest version for your operating system and click **Continue**. See Figure B-1.

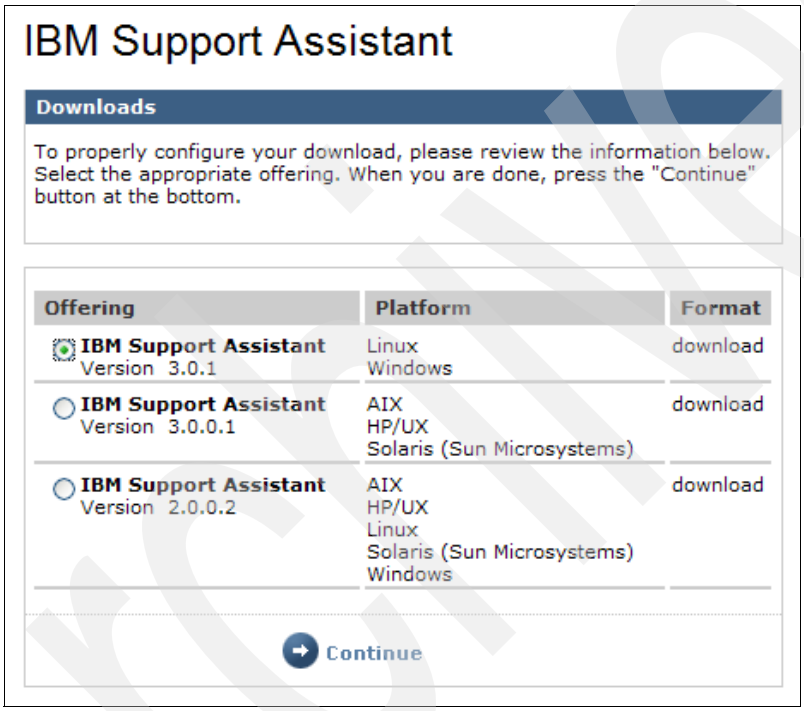


Figure B-1 Selecting the ISA version

4. View the license agreement and select the **I agree** check box to accept it.
5. Click **I confirm**.
6. Select the check box next to your operating system's version and click **Download now**.
7. Click **Save** in the dialog box to save the zip file into a temporary directory on your workstation.
8. When the download of the zip file has completed, unzip the zip files using a zip compression utility (PKZIP, WinZip, or InfoZip) to a temporary directory, for example C:\Downloads\isa_v3.

   Note: Do not unzip the files into a directory name that contains spaces.

9. After it is extracted, run the executable file (setupwin32.exe) to start the installation.

Note: There is a way to install ISA using console or silent mode, but GUI installation is the easiest way for installing ISA.

Refer to the following for detailed information about silent installation:

`<installation_dir>\Installation_and_Troubleshooting_Guide\Installation_and_Troubleshooting_Guide.html`

10. Click **Next** to install ISA to your workstation.
11. View and accept the license agreement, and click **Next**.
12. Specify an installation directory, and click **Next**.
13. Select the features for ISA you want to install, and click **Next**.
14. Review the Summary panel, and click **Install**.
15. When the installation completes successfully, click **Finish**.
16. You must now configure ISA to work with one of the available tools or features. Refer to "Configuring and updating IBM Support Assistant" on page 184.

Configuring and updating IBM Support Assistant

After you have successfully installed ISA you must also download one or more features for the products that you want to use. In our example we have to install WebSphere Application Server V6.1 features to access available JVM tools.

To install a WebSphere Application Server feature, perform the following steps:

1. Start ISA. Refer to "Starting IBM Support Assistant" on page 186 for more information.
2. Click **Updater**.
3. Click the **New Products and Tools** tab as shown in Figure B-2.
4. The plug-in features are categorized by product family. Expand **WebSphere**, and select **WebSphere Application Server V6.1** as shown in Figure B-2.

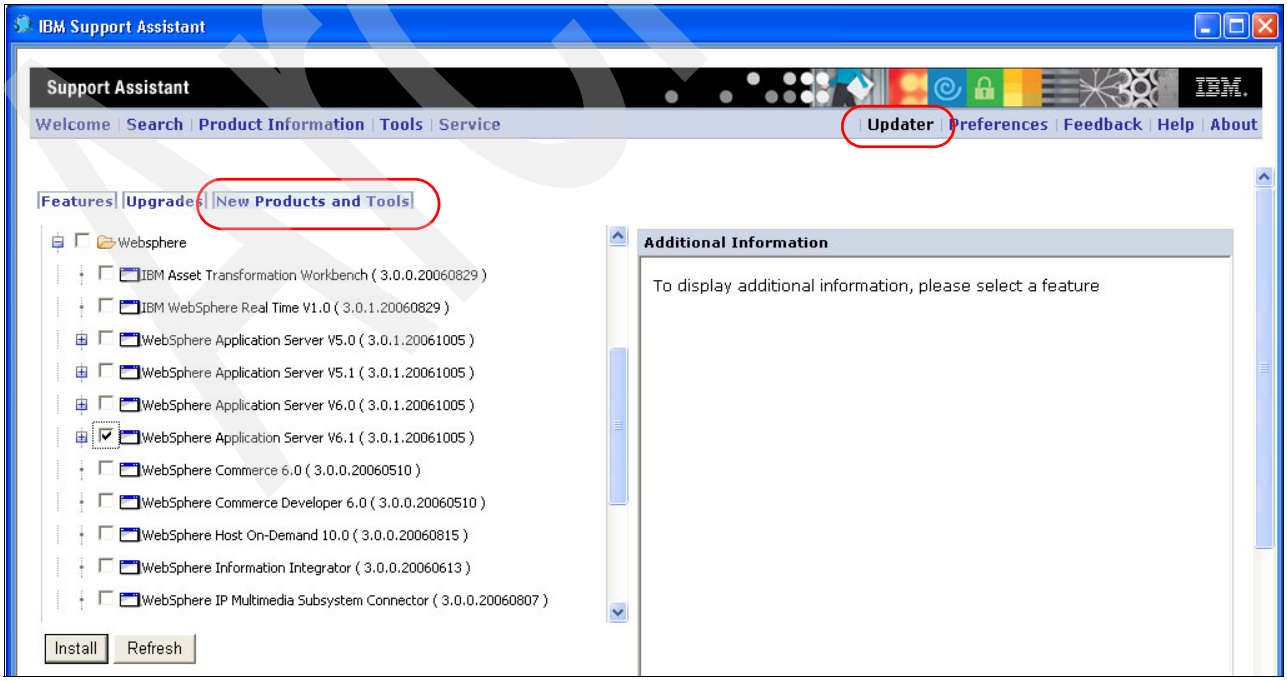


Figure B-2 Installing WebSphere Application Server 6.1 feature

Adding the JVM tools

Next you add the JVM tools:

1. Scroll down the list of available features and expand Common Component Tools as shown in Figure B-3.
2. Select a check box next to the tools that you want to install and click **Install**.

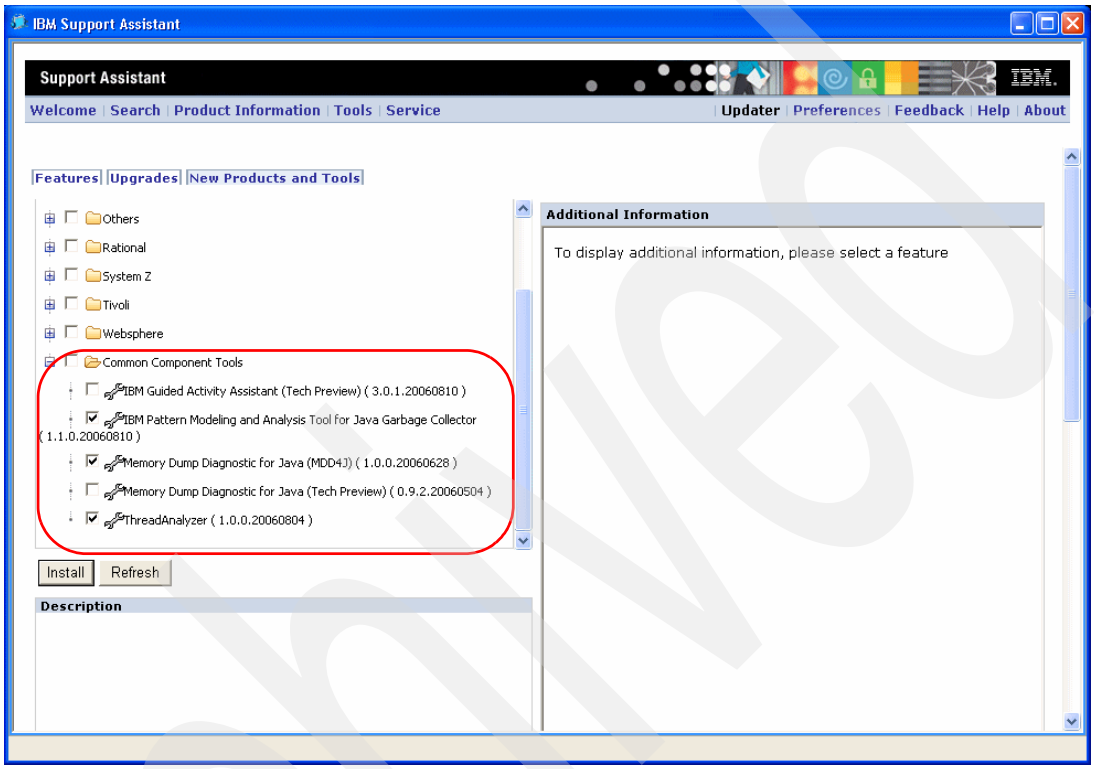


Figure B-3 Adding the JVM tools

3. A new pop-up window is displayed. Be sure to read the license and description for each feature and tool. If you agree with the license, click **I agree**.
4. The installation starts. When all selected tools and features are installed, a pop-up window opens. Click **OK** to restart ISA.

At the time of publication, the following features were available for use with WebSphere Application Server:

- IBM Guided Activity Assistant
- IBM Pattern Modeling and Analysis Tool for Java Garbage Collector (PMAT)
- Memory Dump Diagnostic for Java (MDD4J)
- ThreadAnalyzer

Keeping ISA current is critical to your success in finding the most value out of the ISA tool. Often times, new tools or features are released on an intermittent basis.

Important: It is important to occasionally check for new updates for the installed tools or to see what other tools for your products have been made available since your installation or last update.

Uninstall IBM Support Assistant

You can uninstall ISA in several ways:

- (Windows only) Use the Add/Remove Programs task in the control panel to remove the IBM Support Assistant product.
- (Windows only) Navigate to **Start** → **Programs** → **IBM Support Assistant** → **Uninstall IBM Support Assistant** to start the uninstallation process.
- (All platforms) Navigate to your ISA install location, double-click the **_uninst** folder, and run the uninstall executable file.

Note: You can also uninstall ISA in the console or silent mode. Run the following command from a command prompt:

```
<install_root>\_uninst\<uninstaller_executable_file> -console
```

Or:

```
<install_root>\_uninst\<uninstaller_executable_file> -silent -options
options.txt
```

For example, in Windows:

```
"c:\Program Files\IBM\IBM Support Assistant v3\_uninst\uninstall.exe"
-console
```

Use IBM Support Assistant

Use ISA to search for support information, find product information, analyze product problems using specialized tools, and submit problem reports to IBM that are expedited with automatic data collection.

You can also use the ISA interface to check for new IBM products and tools that are developed for use with ISA.

Starting IBM Support Assistant

On the Windows platforms, ISA starts up with its own GUI windows. On all other platforms, ISA launches the default browser for the system and the ISA application is accessed via the browser interface.

To start ISA on Windows, click **Start** → **All Programs** → **IBM Support Assistant** → **IBM Support Assistant V3**.

To start ISA on other supported platforms, perform the following steps:

1. Open the command prompt.
2. Specify startisa.sh on the command line. A Mozilla browser opens with the IBM Support Assistant V3 welcome page.

Note: If you do not have the Mozilla browser installed, you have to manually open the ISA URL in another supported Web browser. The URL used to access ISA is also written to the console window where you ran the startisa.sh script. The URL is also listed in the <install_root>/workspace/logs/isaurl.log file. Open this file, copy URL, and paste it in a Web browser window.

Starting JVM tools

In order to access any JVM tool, you have to open the WebSphere Application Server V6.1 feature. Follow these instructions to start a JVM tool:

1. In the ISA window click **Tools** in the toolbar.
2. In the left menu click **WebSphere Application Server 6.1**.
3. You must see the list of the installed tools in the right frame. See Figure B-4.

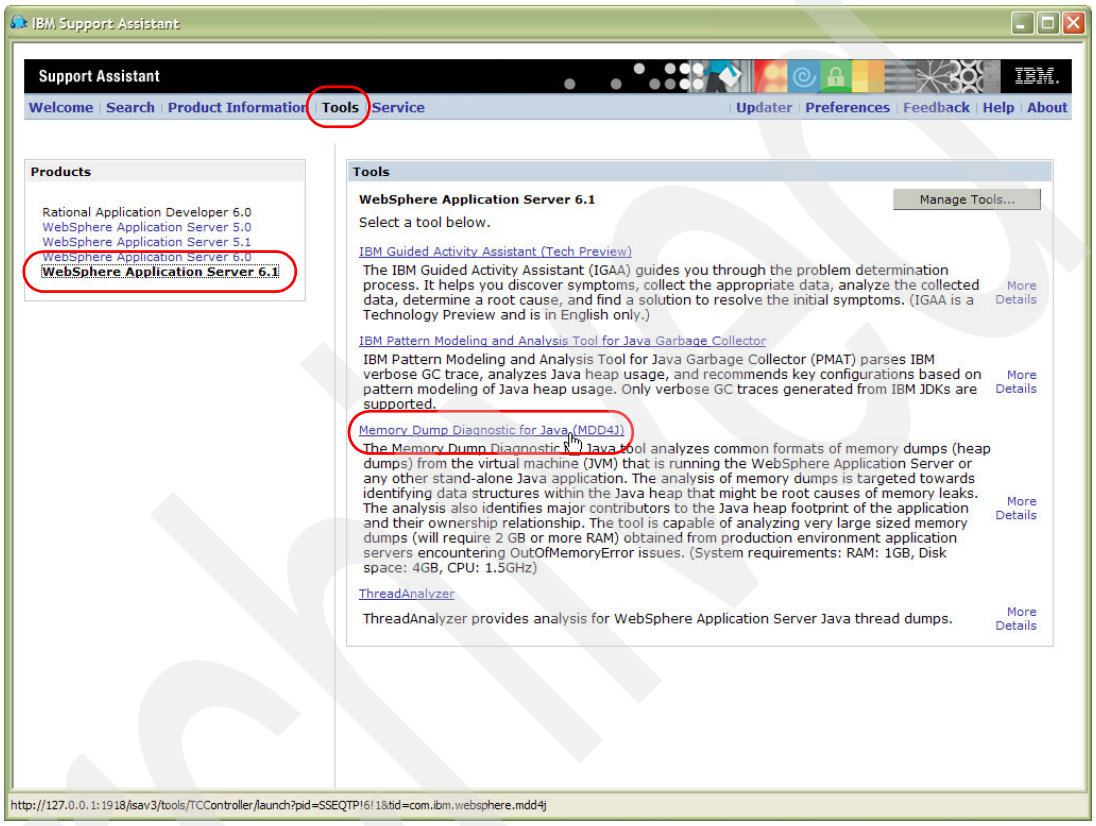


Figure B-4 Available JVM tools

4. Click a link for a desirable tool. In our example we click **Memory Dump Diagnostic for Java (MDD4J)**. If you see any warning message, read its text and click OK. Some tools have prerequisites.
5. A new window opens. Now you can use the tool. Refer to Chapter 8, “Analyzing JVM behavior” on page 83 for the instructions about how to use each tool.

Submitting a problem report to IBM

ISA provides a service feature with an automated system and symptom-based collector. The system collector gathers general information from your operating system, registry, and so on. The symptom-based collection provides the unique ability to collect specific information relating to a particular problem that you are having. It can also provide you with the ability to automatically enable tracing that is helpful to IBM support as part of the data gathering process.

You also can submit problems directly to IBM by creating a problem report and attaching a generated collector file at the same time.

Note: You must use the Portable Collector option to export a .jar file to your i5/OS. The option for Collect Data is not supported on i5/OS.

Perform the following steps to collect data from your i5/OS and submit a problem report with the collected data:

1. Start ISA. Refer to “Starting IBM Support Assistant” on page 186 for more information.
2. Click **Service**.
3. Click **Create Portable Collector**.
4. Select **System collector** in the Select a product box.
5. Enter the integrated file system path to your i5/OS in the Output directory field. For example, as follows:

   `U:\home\collector_output`

   Here `U:` is a mapped network drive to your system’s Integrated File System (IFS) share.
6. Enter the name of the .jar file to be created in the Output file name (*.jar) field. You must enter the .jar extension. For example, as follows:

   `problem.jar`
7. Click **Export**. See Figure B-5.

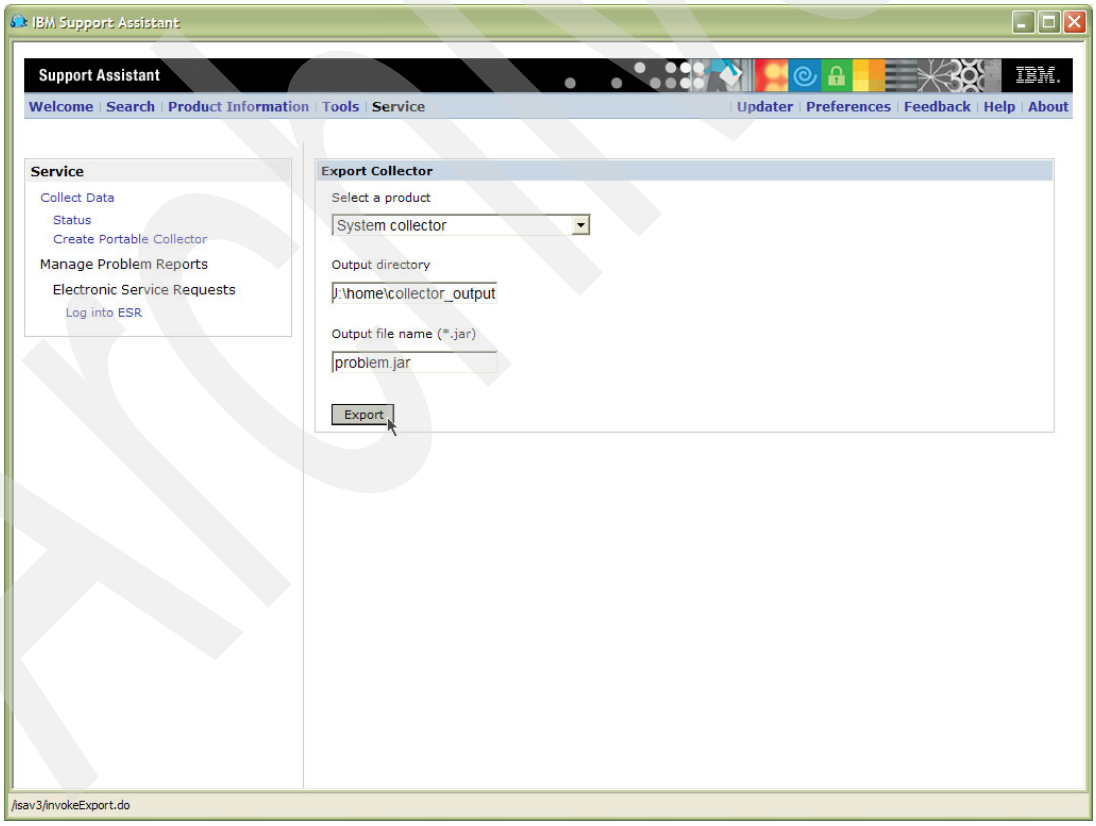


Figure B-5 Creating a portable collector

8. Open an IBM Personal Communications screen and log in to your system.
9. Enter STRQSH on the CL command line and hit Enter.

10. On the QShell command line enter the change directory command to change to your .jar file location. For example:

    ```
    cd /home/collector_output
    ```

11. Enter the following command on the Qshell command line:

    ```
    jar -xvf problem.jar
    ```

    Files are extracted to your directory location.

12. Enter ENV on the Qshell command line to check whether you have a JAVA_HOME variable defined. ISA works best with Java Development Kit (JDK) 1.4 on i5/OS.

13. If you do not have a JAVA_HOME variable defined, enter the following command on the Qshell command line:

    ```
    export JAVA_HOME=/QIBM/ProdData/Java400/JDK14
    ```

    You can verify the JAVA_HOME variable value by running the **java -version** command on the Qshell command line.

14. Enter **chmod -R 755 `find . -name '*.sh'`** on the Qshell command line to make sure your Qshell scripts have execute permission.

15. Enter **./startcollector_iseries.sh** on the Qshell command line to begin system data collection.

16. At the prompt, enter an output file name and path for the newly-created ZIP file. For example:

    ```
    /home/collector_output/12345.724.888.problemoutput.zip
    ```

17. Enter 1 at the prompt to collect system output.

18. The tool prompts you if you want to run another collection. If you are done, type No and hit Enter.

    You now have a ZIP file on your i5/OS in the location you specified. You must now upload it to IBM using ISA.

19. Go back to ISA, and click **Log into ESR**.

20. Enter your IBM ID, Password, IBM customer number, and Country/region of support contract, and click **Login**.

21. On the Submit Problem Report screen, identify your product, select your component, select your contract, provide your relevant information, and attach the ZIP file that was created with the Portable collector tool.

Troubleshooting IBM Support Assistant

There are several log files available in ISA to help with troubleshooting ISA problems.

There is an installation and uninstallation log file available in the location where ISA is installed. The file name is log.txt.

There are log files available for problems starting ISA in the <install_root>/workspace/logs directory. The isa_*.log files represent a set of rolling log files, where the most recent log file is always named isa_0.log.

There is also a log file in the <install_root>/workspace/.metadata directory that contains logging messages that might occur either before ISA logging is initialized or if logging initialization fails.

Refer to the following file for some troubleshooting information:

```
<ISA_installation_files_dir>\Installation_and_Troubleshooting_Guide\en\Installatio
n_and_Troubleshooting_Guide_en.html
```

This file is part of the directory that is created when you expand the ZIP file downloaded from the Web. Refer to “Downloading IBM Support Assistant from Web” on page 183.

C

Diagnostic Tools and Framework for Java Dumps

This section provides information about the Diagnostic Tools and Framework for Java Dumps (DTFJ), which has been enhanced in version 5.0 of IBM Technology for Java virtual machine (JVM). DTFJ is a flexible framework that allows diagnostics tools to be written in a platform, runtime, and version-independent manner.

This appendix covers the “Overview of DTFJ” on page 192.

Overview of DTFJ

DTFJ has two main abstract entities at its foundation, an *image* and *runtime environment*. An image is any complete operating system image, such as a system dump file. A runtime environment is any language-specific runtime environment, such as a JVM. DTFJ provides a layer of abstraction, making the image format and internal data structures of the runtime environment transparent. Because of this abstraction, DTFJ can be extended to encompass other operating system images and languages, although we focus on the JVM in this section. DTFJ could be extended to handle diagnostic data from other vendors' JVMs in the future also.

One of the core components of DTFJ is an application programming interface (API) to provide access to data stored in the operating system image, making it simpler to write Java diagnostics tools. In the implementation of the DTFJ API shipped with IBM Technology for JVM, the basic abstract concepts of an image and runtime environment have been extended by adding interfaces for Java-specific entities such as Java threads.

Tools that use the DTFJ API can gain access to the huge array of information about a Java process that is available in a system dump. No knowledge of the system dump format or how Java objects/structures are laid out in memory is required by a tool-writer.

Using the DTFJ API, cross-platform tools can be written in Java to process the wide range of data available in system dump files. This includes information about the platform on which the process is running, including:

- Physical memory
- CPU number and type
- Libraries
- Commands
- Thread stacks
- Registers

The API can also provide information about the state of the Java runtime and the Java application it is running, including:

- Class loaders
- Threads
- Monitors
- Heaps
- Objects
- Java threads
- Methods
- Compiled code
- Fields and their values

Because you can create system dump files non-destructively (for example, using the -Xdump command-line option or calling the static method com.ibm.jvm.Dump.SystemDump()), you can create tools to perform analysis of system dumps collected while the JVM in question is still running, in addition to traditional analysis of system dumps generated due to a JVM crash.

jextract

Due to the differences in system dump format between platforms and Java releases, the jextract utility must run against a system dump file before it can be processed using any tool that utilizes the DTFJ API methods. jextract must be run on the system that produced the system dump. jextract handles the JVM-version-specific and platform-specific details for you,

producing metadata from the system dump that allows the internal structure of the JVM to be analyzed. The output from jextract can be passed to a tool that uses the DTFJ API.

jextract is supplied with IBM Technology for JVM in the following directory:

```
/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/bin
```

You supply the name of the input system dump file as a command-line argument to jextract. For example, issuing the following command in a Qshell session:

```
jextract /home/rumsbya/core.20061002.140109.9288.dmp
```

This command runs jextract on the system dump file core.20061002.140109.9288.dmp. Because no output filename is specified, jextract simply adds a .zip extension to the input filename when creating the output file. The output zip file or files contained in it are then typically passed as input to tools built on the DTFJ API.

Tools based on DTFJ API

The DTFJ API is intended to facilitate the writing of tools to analyze the variety of dump files that the JVM can create. Such tools are anticipated to be developed by both IBM and interested parties who require a customized tool for their specific requirements. Because the DTFJ itself is relatively new, there are a limited number of tools available at the time of writing that exploit the API. However, the following tools available through the IBM Support Assistant use or shortly are going to use the DTFJ API:

- ThreadAnalyzer
- MDD4J
- EVTK

The DTFJ allows speedier development of further diagnostics tools due to the common API and planned re-use of some common tool components. It is worth mentioning that a tool's usage of the DTFJ API is transparent to end users of the tool. Of course, as improvements to the diagnostics tools are made and new features are implemented, undoubtedly the tool interfaces are going to be updated to reflect these.

DTFJ API documentation

The DTFJ API documentation is shipped as a zip file with IBM Technology for JVM on most platforms. i5/OS users can obtain this zip file by visiting the FTP site for this IBM Redbook. Refer to Appendix E, "Additional material" on page 199 for more information about accessing the FTP site. The DTFJ API documentation is available in the DTFJ folder of the extracted apidoc.zip file.

For guidance on how to use the DTFJ API for tool development, refer to the following IBM developerWorks article:

http://www-128.ibm.com/developerworks/java/library/j-ibmjava5/

We also suggest consulting the IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 Diagnostics Guide for more information about DTFJ:

http://www-128.ibm.com/developerworks/java/jdk/diagnosis/

DTFJ API or JVMTI?

For tool writers, the DTFJ API presents another option for accessing information about the state of a JVM. There naturally may arise a question as to what distinguishes the DTFJ API available in IBM Technology for JVM from tooling mechanisms provided in the Java 2 software development kit (SDK), such as the JVM Tools Interface (JVMTI).

The DTFJ API and JVMTI have different goals and implementations. Therefore, although there is a degree of crossover in terms of the data that can be obtained through these mechanisms, the way in which the data is gathered and used is markedly different in most cases. It is not intended in any way for the DTFJ API to replace the JVMTI because they are intended for different purposes.

Table C-1 presents the major differences between the DTFJ API and JVMTI.

Table C-1 Differences between the DTFJ API and JVMTI

	DTFJ API	**JVMTI**
Interface type	Java based on extendable API	Native code, based on JVM callbacks
Typical use	Offline analysis of system dumps; troubleshooting	Live monitoring and control of JVM; profiling an application
Tool format	Java application running in separate JVM	Native library (agent) specified on Java command line
Tool portability	Java code can be run on any compatible JVM on any platform	Native code has to be recompiled for different platforms
Provides means to instrument application?	No	Yes
Live or post-mortem analysis?	Tool can run on system dump gathered non-destructively from live JVM or generated automatically in the event of JVM failure	Tool runs while the JVM being analyzed is live
Impact on JVM being analyzed	For post-mortem analysis, none. Taking a non-destructive system dump can take time. The tool runs in a separate JVM, therefore, no impact after the system dump is generated	Variable. Depends on what type of events the agent has asked to be notified about
Maturity	Since IBM JDK 1.4.2	Since J2SE 5.0. Predecessor JVMPI present since Java 1.1
Applicability	Unique to IBM JVMs at present	JSR-163 compliant, therefore, supported across all J2SE 5.0-compliant JVMs, including IBM Technology for JVM

D

jconsole

jconsole is a standard tool to monitor runtime parameters of the Java virtual machine (JVM). This appendix demonstrates how to start jconsole from your workstation. During the time of writing this book jconsole, however, is not supported in WebSphere Application Server.

Monitoring IBM Technology for JVM remotely with a GUI

This section shows you how to use the Java Management Extensions (JMX™) utility to monitor the IBM Technology for JVM. For this example we use the jconsole utility that is included with J2SE 5.0 software development kit (SDK).

There are two basic steps involved:

1. Initialize the JVM that is going to be monitored.

   For demonstration purposes this example has Secure Sockets Layer (SSL) and authentication disabled. For details about using security with JMX, see the following link:

   http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html#remote

   We add the following command line arguments to our **java** invocation:

   ```
   -Dcom.sun.management.jmxremote.port=44444
   -Dcom.sun.management.jmxremote.authenticate=false
   -Dcom.sun.management.jmxremote.ssl=false
   ```

   For example:

   ```
   java -Dcom.sun.management.jmxremote.port=44444
   -Dcom.sun.management.jmxremote.authenticate=false
   -Dcom.sun.management.jmxremote.ssl=false -cp ../jars/bughunt.jar
   bughunt.hanger.Eatery
   ```

2. Connect the jconsole utility to the running process.

   Find the jconsole utility on your workstation. It must be in <JDK_INSTALL>\bin, where JDK_INSTALL is the directory to which the JDK was installed. For example:

   ```
   C:\Program Files\Java\jdk1.5.0_06\bin
   ```

   Open a Microsoft Windows command prompt and run the command similar to the following:

   ```
   C:\Program Files\IBM\Java50\bin>jconsole <system_hostname>:44444
   ```

   Here <system_hostname> is a host name of the server with active JVM that you have started in the previous step. The window shown in Figure D-1 must open.

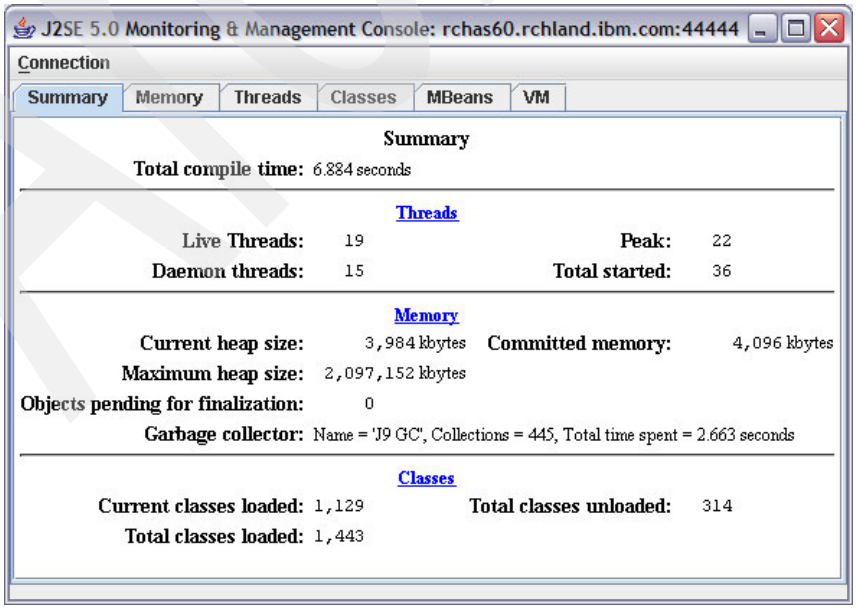


Figure D-1 Monitoring and Management console (jconsole)

Each tab has some useful information. Figure D-2 shows an example of what you see on the threads tab.

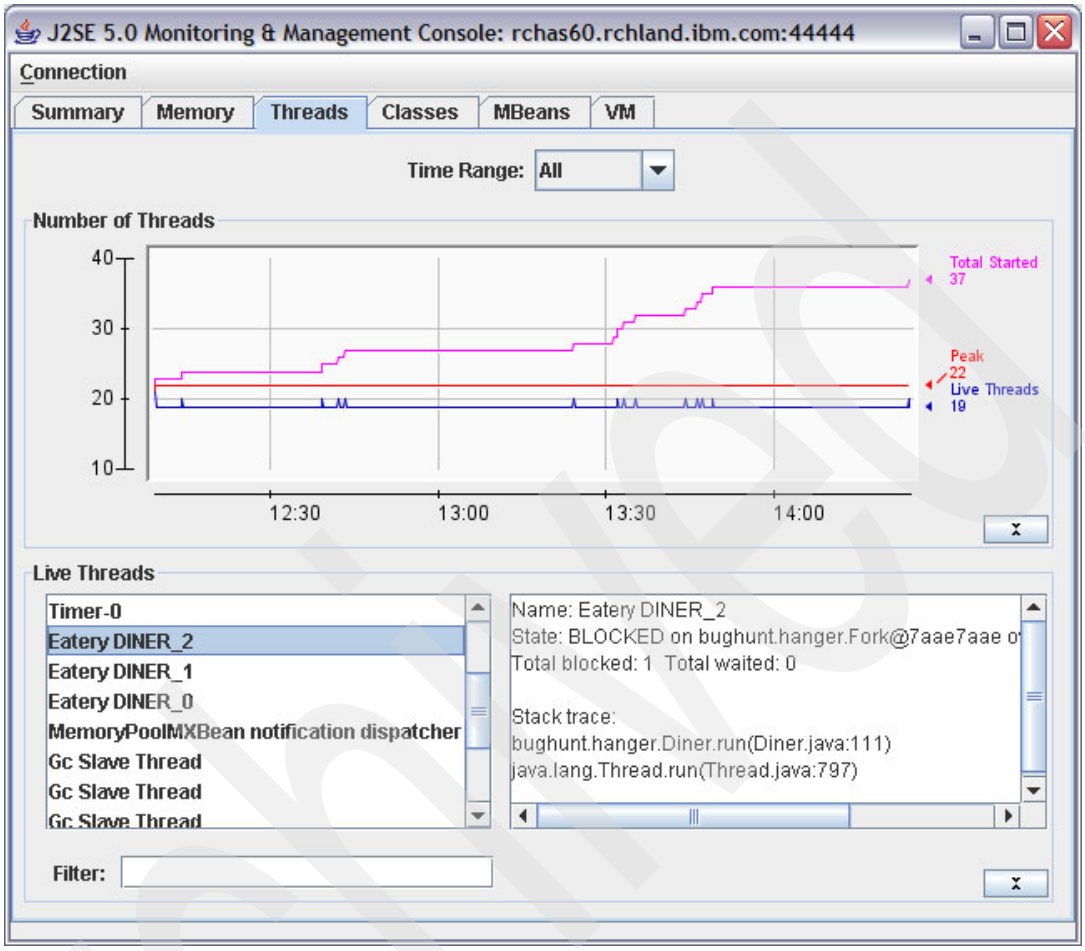


Figure D-2 The threads tab

Explore other parameters by selecting other tabs.

E

Additional material

This IBM 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 IBM 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/SG247353

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 IBM Redbook form number, SG247353.

Using the Web material

The additional Web material that accompanies this IBM Redbook includes the following files:

File name	*Description*
apidoc.zip	This file contains Javadoc™ for several APIs

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder. Switch to any subfolder under *apidoc* folder, for example DTFJ, and open the *index.html* file from this folder in a Web browser.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this IBM Redbook.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 201. Note that some of the documents referenced here may be available in softcopy only.

- *WebSphere Application Server V6 Problem Determination for Distributed Platforms*, SG24-6798

Online resources

These Web sites are also relevant as further information sources:

- IBM eServer iSeries Information Center

  http://publib.boulder.ibm.com/iseries/
- IBM(R) Developer Kit and Runtime Environment, Java(TM) 2 Technology Edition, version 5.0, Diagnostics Guide:

  http://publib.boulder.ibm.com/infocenter/javasdk/v5r0/index.jsp
- IBM Systems Workload Estimator

  http://www.ibm.com/systems/support/tools/estimator/index.html
- Java Native Interface documentation

  http://java.sun.com/j2se/1.5.0/docs/guide/jni/index.html
- Performance Management for IBM System i

  http://www-03.ibm.com/servers/eserver/iseries/perfmgmt/resource.html
- System i group PTFs by release

  http://www-912.ibm.com/s_dir/slineOO3.NSF/GroupPTFs?OpenView&view=GroupPTFs
- WebSphere Application Server V6.1 Information Center

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp

How to get IBM Redbooks

You can search for, view, or download IBM Redbooks, IBM Redpapers, Hints and Tips, draft publications, and additional materials, and also order hardcopy IBM Redbooks or CD-ROMs, at the following Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

H

I

J

K

L

M

N

O

P

Q

X

IBM Technology for Java Virtual Machine in IBM i5/OS

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

IBM Technology for Java Virtual Machine in IBM i5/OS

Redbooks

Most information about 32-bit JVM applies to all IBM server platforms

Comprehensive book on new 32-bit JVM in i5/OS

Run WebSphere Application Server in 32-bit JVM

This IBM Redbook gives a broad understanding of a new 32-bit Java Virtual Machine (JVM) in IBM i5/OS. With the arrival of this new JVM, IBM System i platform now comfortably supports Java and WebSphere applications on a wide array of different server models: from entry size boxes to the huge enterprise systems.

This book provides in-depth information about setting Java and IBM WebSphere environments with new 32-bit JVM, tuning its performance, and monitoring or troubleshooting its runtime with the new set of tools.

Information in this IBM Redbook helps system architects, Java application developers, and system administrators in their work with 32-bit JVM in i5/OS.

Important: Despite the fact that this book targets i5/OS implementation, most information in this book applies to all IBM server platforms, where the new 32-bit JVM is supported.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-7353-00 ISBN 0738486051