WebSphere®

IBM

WebSphere Real Time User Guide

Note
Before using this information and the product it supports, be sure to read the general information under "Notices" on page 279.

First edition (September 2006)

This edition of the User Guide applies to IBM WebSphere Real Time, Version 1, and to all subsequent releases and modifications until otherwise indicated in new editions.

Preface

This User Guide provides general information about IBM WebSphere Real Time.

Contents

Chapter 1. Introduction

This information center describes the IBM WebSphere Real Time product referred to as WebSphere Real Time in this information.

You can use this information to install and configure WebSphere Real Time. Selected information on non-Real Time Java is also provided here. Further diagnostics information can be found in the Diagnostics Guide.

A sample application is included which is converted from a standard Java™ application into a real-time application.

For the latest information see the readmeFirst.txt file in the docs directory.

Who should read this information

There are three groups of readers for this information:

- Java application programmers who develop WebSphere Real Time applications; see Table 1.
- System administrators who install and configure the Java environment; see Table 2.
- Service personnel team who maintain and develop the Java environment; see Table 3 on page 2.

Table 1. Application programmers' tasks

Task	Reference
Planning your application.	"Planning your WebSphere Real Time application" on page 50
Running a standard Java application in real time without any changes to the process or the code.	"Running the sample application with Metronome" on page 44
Running a standard Java application in real time but avoiding the unpredictable time delays caused by JIT.	"Running the sample application using AOT" on page 45 Chapter 3, "Using the ahead-of-time compiler and WebSphere Real Time," on page 15
Writing a Java application that has a more predictable time control.	"Writing real-time threads" on page 52
Writing a Java application that can respond to external events.	"Writing asynchronous event handlers" on page 54
Writing Java application components with sub-millisecond precision that cannot tolerate any delay due to garbage collection activities.	"Writing NHRT threads" on page 56

Table 2. System administrators' tasks

Task	Reference
Planning for and overseeing product installation.	Chapter 2, "Installing WebSphere Real Time," on page 7

Table 2. System administrators' tasks (continued)

Task	Reference
Tuning the real-time environment.	"Tuning Metronome Garbage Collector" on page 27

Table 3. Service personnel tasks

Task	Reference
Troubleshooting system and performance problems.	Chapter 9, "Troubleshooting in a real-time environment," on page 69
Diagnosing problems.	"First steps in problem determination" on page 79

Overview of WebSphere Real Time

WebSphere Real Time bundles real-time capabilities with the standard JVM.

You enable real-time capabilities by using the **-Xrealtime** option when running the JVM or any of the tools provided. By default, the JVM and the tools provided run without real-time capabilities enabled. Figure 1 on page 3 shows the relationships of the two JVMs that are supplied with WebSphere Real Time.

The following Java commands recognize the **-Xrealtime** option:

Table 4. Java commands used in real-time mode

Command	Function
java	Runs in standard mode by default but also runs in real-time mode when the **-Xrealtime** option is specified. Real-time mode lets the programmer access classes from the javax.realtime package. You can use precompiled jar files from the jxeinajar tool and the Metronome deterministic garbage collection technology.
javac, javah, javap	Runs in standard mode by default; but when the **-Xrealtime** option is specified, it includes the javax.realtime.* classes in the classpath.
jxeinajar	Can be run only in real-time mode. Running jxeinajar without the **-Xrealtime** option is not supported.
jdmpview	Runs in standard mode by default but also runs in real-time mode when the **-Xrealtime** option is specified. The syntax and mode of operation is entirely different when run in the real-time mode. When run in real-time mode, **jdmpview -Xrealtime** in should be run in conjunction with **jextract -Xrealtime**.

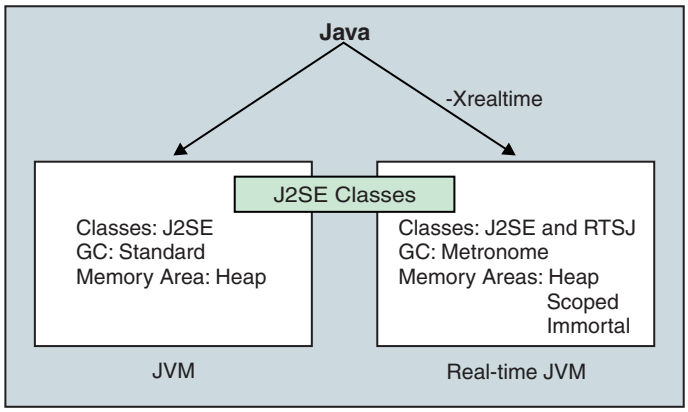

Figure 1. Overview of WebSphere Real Time

Features of WebSphere Real Time

Real-time applications need predictable timing and repeatable execution duration, rather than absolute speed.

When the JVM is run with the **-Xrealtime** option, the JVM has two additional memory heaps; one for scoped and the other for immortal memory. It also uses the Metronome Garbage Collector to achieve time-based collections. When the JVM is run in a traditional throughput (non-Xrealtime) mode without the **-Xrealtime** option, various work-based garbage collectors can be used that optimize throughput but can have larger individual delays than the Metronome Garbage Collector.

The main concerns when deploying real-time applications with traditional JVMs are:
- Unpredictable (potentially long) delays from Garbage Collection (GC) activity.
- Delays to method execution as Just-In-Time (JIT) compilation and recompliation occurs, with variability in execution time.
- Arbitrary operating system scheduling.

WebSphere Real Time removes these obstacles by providing:
- The Metronome Garbage Collector, an incremental, deterministic garbage collector with very small pause times
- Ahead-Of-Time (AOT) compilation
- Priority based FIFO scheduling

In addition, WebSphere Real Time provides the real-time programmer with the RTSJ facilities - see Chapter 5, “Support for RTSJ,” on page 29.

Benefits

The benefits of the real-time environment are that Java applications run with a greater degree of predictability than with the standard JVM and provide a consistent timing behavior for your Java application. In this sense, background activities, such as compilation and garbage collection, occur at given times and thus remove the occurrence of any unexpected peaks of background activity during an application’s execution.

You obtain these advantages by extending the JVM with the following functions:

- Metronome real-time garbage collection technology
- Ahead-of-time (AOT) compilation
- Support for the Real-Time Specification for Java (RTSJ)

All Java applications can be run in a real-time environment without modification, benefiting from the Metronome Garbage Collector and its deterministic garbage collection that occurs at regular intervals. To achieve the maximum benefit from WebSphere Real Time, you can write applications specifically for the real-time environment using both real-time threads and no-heap real-time threads. The approach that you take depends on the timing specification of your application.

Many real-time Java applications can to exploit the Metronome Garbage Collector's low pause times and AOT to achieve their goals, retaining the benefits of Java portability. For applications with tighter requirements, using the RTSJ facilities of real-time threads and no-heap real-time threads, in conjunction with scoped and immortal memory could be used. This approach limits your application to run in a real-time environment losing the advantage of portability to JSE Java. It also means that you have to develop a more complex programming model.

Considerations

There are a number of factors that you should be aware of when using WebSphere Real Time.

They are:

- Running two real-time JVMs on the same machine is very complex. There would be two garbage collectors and each JVM does not know about the memory areas of the other. Therefore neither JVM can know what is feasible.
- The **-Xshareclasses** option is not supported with **-Xrealtime** because no guarantees can be given when multiple JVMs are sharing resources.
- You cannot use the **-Xdebug** option and the **-Xnojit** option with code that has been precompiled using the ahead-of-time compiler because **-Xdebug** compiles code in a different way from the Ahead-of-Time (AOT) compiler and is not supported.

  To debug your code, use interpreted or JIT-compiled code.
- If you are using the com.sun.tools.javac.Main interface to compile Java source code that uses the javax.realtime package, you will need to ensure that sdk/jre/bin/realtime/jclSC150/realtime.jar is included in the classpath. One common example of this type of compilation is ant compilation.

Performance considerations

WebSphere Real Time is optimized for predictable execution times and short GC pauses rather than the highest throughput performance or smallest memory footprint.

Achieving tight temporal characteristics and support for the Real-Time Specification for Java (RTSJ) required some standard IBM Java runtime optimizations to be disabled. Consequently, a reduction in overall performance is likely to be observed when a standard Java application is run with the **-Xrealtime** parameter.

Reducing variability

The two main sources of variability in a standard JVM are handled in the WebSphere Real Time as follows:

- Java code preparation: loading and Just-In-Time (JIT) compilation is dealt with by Ahead-Of-Time (AOT) compilation. See "Using the AOT compiler" on page 16.
- Garbage Collection pauses: the potentially long pauses from standard Garbage Collector modes are avoided by using the Metronome Garbage Collector. See Chapter 4, "Using the Metronome Garbage Collector," on page 21.

Migration

WebSphere Real Time runs in a customized Linux® environment. You can use standard Java applications in a real-time environment. Alternatively, you can modify your applications to exploit the new features of WebSphere® Real Time.

System migration

Follow the instructions provided by the Linux support team.

Chapter 2. Installing WebSphere Real Time

Start here to follow the sequence of steps to ensure the successful installation of the product.

Product deliverables

The installation file for IBM WebSphere Real Time is called ibm-ws-rt-sdk-1.0-1.0-linux-i386.tgz .

The IBM® Eclipse Help system, version 3.1.1. This help system can be installed locally and the WebSphere Real Time documentation plug-in jar file can be copied to the plug-in directory.

Software and hardware prerequisites

Use this list to check the operating system, Java environment, and hardware that are supported for WebSphere Real Time.

This product can be used with the following operating systems and hardware.

Operating system

- A customized version of Linux X86 on Red Hat EL4 with extensions. See "Customizing your Linux environment" on page 9.
- Red Hat Enterprise Linux 4 Update 2, using a modified 2.6.16 base kernel and glibc.

Hardware

The configurations supported for WebSphere Real Time are multiprocessor variants of the following systems:.

- IBM Blade Server LS20 (Type 8850)
- IBM eServer™ 326m (Type 7969)
- IBM Intellistation A Pro (Type 6217)
- IBM Blade Server LS21 (Type 79716AU)

Viewing the information center

In the docs directory, you will find: com.ibm.rt.doc.jar, com.ibm.rt.doc.zip, and rt_jre.pdf files.

- com.ibm.rt.doc.jar can be copied to the plugin directory of either your Eclipse Help System version 3.1.1 or to the plugin directory of Eclipse SDK version 3.1.2 or later.
- com.ibm.rt.doc.zip can be unpacked into the plugin directory of your Eclipse Help System if the version is earlier than 3.1.1.
- The rt_jre.pdf file uses Adobe Acrobat.

To use the information center on your local machine, you must install the Eclipse Help system; see "Viewing the online help" on page 11. You can also copy the jar file to the plugin directory of Eclipse SDK and view the information center using **Help → Help Contents**.

Useful tools

These tools can be used in association with WebSphere Real Time. Some of these tools are in the early stages of development and may not be fully supported.

For application development:

Table 5.

Product	Download from:
Eclipse SDK 3.1.2 or later provides a complete application development environment for real-time applications. You can also copy the information center file, com.ibm.rt.doc.jar, to the plugin directory to view the documentation associated with this product.	http://www.eclipse.org/downloads/

Miscellaneous tools and enhancements:

Table 6.

Product	Download from
Information about the latest tools that you can use to support WebSphere Real Time, for example: Tuning fork, eventrons and real-time class analysis tool (ratcat).	http://www.alphaworks.ibm.com/topics/realtimejava

Customizing your Linux environment

To use WebSphere Real Time, refer to the Real-Time Linux installation information.

Use the following documents supplied with your Real Time Linux Installation CD to set up the correct environment for WebSphere Real Time.

- *Installation instructions for RT-Linux RHEL 4 / Update 2 build on LS 20 / HS 20 Blades for IBM Blade Center* in the file linux-base-install.txt.
- *Real Time Linux Installation* in rtlinux-install.txt.
- *Real Time Linux Build notices* in rtlinux-build.txt.
- *Release Notes for Real Time Linux* in release-notes.txt

Unpacking the WebSphere Real Time gzipped tar file

The JVM is supplied in a gzipped file called ibm-ws-rt-sdk-1.0-1.0-linux-i386.tgz. It can be installed into any directory.

To unpack the Java driver, follow these instructions:

1. From a shell prompt, enter: `useradd -G realtime -m username`. This command adds the *username* to the realtime group. Without this access you will lack the capabilities to do things like request SCHED_FIFO locked memory. If you are not a member of the realtime group, the following message is issued.

   ```
   Error: Port Library failed to initialize  Could not create the Java virtual machine.
   ```

   To add realtime as a supplementary group to an existing user, substitute your login name for *username*. To make this substitution, you need superuser authority.
2. From a shell prompt, enter:

   ```
   tar xzf ibm-ws-rt-sdk-1.0-1.0-linux-i386.tgz -C target_directory
   ```

   where *target_directory* is your working directory. This command creates the following directories:

   ```
   java2-i386-50/
                       bin/
                       COPYRIGHT
                       demo/
                       docs/
                       include/
                       jre/
                       lib/
                       realtime.src.jar
                       src.jar
   ```

See “Testing your installation” on page 11 to verify that your installation has been successful.

Setting the PATH

After setting the **PATH** environment variable, you can run an application or program by typing its name at a shell prompt with a filename as an argument.

Note: If you alter the **PATH** environment variable as described below, you override any existing Java executables in your path.

You can specify the path to a tool by typing the path before the name of the tool each time. For example, if the SDK is installed in /opt/ibm-java2-i386-50/jre/bin, you can compile a file named *myfile.java* by typing the following at a shell prompt:

```
/opt/ibm/java2-i386-50/bin/javac myfile.java
```

To avoid typing the full path each time:

1. Edit the shell startup file in your home directory (usually .bashrc, depending on your shell) and add the absolute paths to the **PATH** environment variable; for example:

   ```
   export PATH=/opt/ibm/java2-i386-50/jre/bin:/opt/ibm/java2-i386-50/bin:$PATH
   ```

2. Log on again or run the updated shell script to activate the new **PATH** setting.
3. Compile the file with the javac tool. For example, to compile the file *myfile.java*, at a shell prompt, enter:

   ```
   javac -Xrealtime myfile.java
   ```

   The **PATH** environment variable enables Linux to find executable files, such as javac, java, and javadoc, from any current directory. To display the current value of your **PATH**, type the following at a command prompt:

   ```
   echo $PATH
   ```

Setting the CLASSPATH

The **CLASSPATH** environment variable tells the SDK tools, such as java, javac, and javadoc, where to find the Java class libraries.

Set the **CLASSPATH** explicitly only if one of the following applies:

- You require a different library or class file, such as one that you develop, and it is not in the current directory.
- You change the location of the bin and lib directories and they no longer have the same parent directory.
- You plan to develop or run applications using different runtime environments on the same system.

To display the current value of your **CLASSPATH**, enter the following at a shell prompt:

```
echo $CLASSPATH
```

If you develop and run applications that use different runtime environments, including other versions that you have installed separately, you must set **CLASSPATH** and **PATH** explicitly for each application. If you run multiple applications simultaneously and use different runtime environments, each application must run in its own shell.

If you run only one version of Java at a time, you can use a shell script to switch between the different runtime environments.

Testing your installation

Use the **-version** option to check if your installation is successful.

The Java installation consists of a standard JVM and a real-time JVM.

Test your installation by following these steps:

1. At a shell prompt, enter `java -version`

   This command returns the following messages if it is successful:

```
java version "1.5.0"
Java(TM) 2 Runtime Environment, Standard Edition (build pxi32rt23-20060809b)
IBM J9 VM (build 2.3, J2RE 1.5.0 IBM J9 2.3 Linux x86-32 j9vmxi3223ifx-20060719 (JIT enabled)
J9VM - 20060714_07194_lHdSMR
JIT  - 20060428_1800.ifix2_r8
GC   - 200607_07)
JCL  - 20060809
```

   Note: The version information is correct but the dates will be later than this example. The format of the date string is: yyyymmdd followed possibly by additional information specific to the component.

2. At a shell prompt, enter `java -Xrealtime -version`

   This command returns the following messages if it is successful:

```
java version "1.5.0"
Java(TM) 2 Runtime Environment, Standard Edition (build pxi32rt23-20060809b)
IBM J9 VM (build 2.3, J2RE 1.5.0 IBM J9 2.3 Linux x86-32 j9vmxi32rt23-20060809a (JIT enabled)
J9VM - 20060809_07538_lHdRRr
JIT  - 20060802_2345_r8.rt
GC   - 200608_02-Metronome
RT   - GA_2_3_RTJ--2006-07-24-AA-IMPORT)
JCL  - 20060809
```

   Note: The version information is correct but the dates will be later than this example. The format of the date string is: yyyymmdd followed possibly by additional information specific to the component.

Viewing the online help

In the docs directory, the documentation is provided for use in the Eclipse help system as com.ibm.rt.doc.jar and com.ibm.rt.doc.zip. The information is also provided as an Adobe PDF file called rt_jre.pdf.

- com.ibm.rt.doc.jar can be copied directly into the plugin directory of your Eclipse Help System version 3.1.1 or the plugin directory of Eclipse SDK 3.1.2 or later.
- com.ibm.rt.doc.zip can be unpacked into the plugin directory of your Eclipse Help System if the version is earlier than 3.1.1.
- rt_jre.pdf is for use with Adobe Acrobat.

To use the information center on your local machine, you need to install the Eclipse Help system.

1. Installing the Eclipse Help System.
   a. Download the latest version of the Eclipse Help System version 3.0.1 from http://www.alphaworks.ibm.com/tech/iehs/download.
   b. Select the zip, tar, or tgz file that is appropriate for your operating system.
   c. Create a new directory where you plan to install the Eclipse Help System. This is referred to as <INSTALL_DIR> in the rest of this document.

d. Unpack the file into, for example, /opt/<INSTALL_DIR> or C:\<INSTALL_DIR> directory depending on your operating system. This creates a directory called /opt/<INSTALL_DIR>/ibm_help on Linux or C:\<INSTALL_DIR>\IBM_Help_301_Win\ibm_help on Windows.

2. Adding the WebSphere Real Time Information Center to your Eclipse Help System for versions earlier than 3.1.1.
   a. Extract the files from com.ibm.rt.doc.zip into the /opt/<INSTALL_DIR>/ibm_help/eclipse/plugins directory on Linux or C:\<INSTALL_DIR>\IBM_Help_301_Win\ibm_help\eclipse\plugins on a Windows® system.
   b. To start the Eclipse Help System, change directory to /opt/<INSTALL_DIR>/ibm_help and enter `help_start`.
3. Adding the WebSphere Real Time Information Center to your Eclipse Help System for version 3.1.1 or later only. You can also so this if you are using Eclipse SDK 3.1.2 or later.
   a. Copy com.ibm.rt.doc.jar to the plugin directory in the help system. For example, this directory is /opt/<INSTALL_DIR>/ibm_help/eclipse/plugins directory on Linux or C:\<INSTALL_DIR>\IBM_Help_301_Win\ibm_help\eclipse\plugins on a Windows system.
   b. To start the Eclipse Help System, change directory to the /opt/<INSTALL_DIR>/ibm_help directory and enter `help_start`.
4. Customizing the Eclipse Help System. If you want to make the WebSphere Real Time Information Center the default when you start your Eclipse Help System.
   a. Edit the help_start.sh or help_start.bat file and change the **-plugincustomization** option to:

      ```
      -plugincustomization plugins\com.ibm.rt.doc\plugin_customization.ini
      ```

      The command line in the help_start.bat file looks like this:

```
.\jre\bin\java -classpath eclipse\plugins\org.eclipse.help.base_3.1.0
     \helpbase.jar org.eclipse.help.standalone.Help
   -eclipsehome eclipse   -plugincustomization plugins\com.ibm.rt.doc\plugin_customization.ini
   -command displayHelp -host 127.0.0.1 -showupdater  -noexec
```

5. Closing the Eclipse Help System. When you have finished with the help system, enter `help_end`. Otherwise, the next time you try to start the system, you will not be able to start the system because of a running process.
6. Running the Eclipse Help System.
   a. Using the search function. The first time you search, the search pauses while indexing takes place.
   b. Filtering your searches. You can ″Set Scope″ so that it only searches the WebSphere Real Time Information Center. Follow the prompts.
   c. Printing. From the navigation tree, click the icon that appears when you hover over a topic in the navigation tree. The pop-up allows you to select that topic or all of the sub-topics associated with that topic. Click your preference and a new window opens allowing you to confirm the selection you want to print that part. Submit the job to your local printer in the usual way.
   d. Installing on a Local Area Network. Refer to the release notes that come with the Eclipse Help System.
   e. Using a CD. Refer to the release notes that come with the Eclipse Help System.

The information center is also provided as PDF. The information has not been fully optimized for PDF.

Uninstalling the SDK and Runtime Environment for Linux

A list of the steps to remove WebSphere Real Time SDK and Runtime Environment for Linux.

1. Remove the Runtime Environment or Runtime Environment files from the directory in which you installed the SDK or Runtime Environment.
2. Remove from your **PATH** statement the directory in which you installed the SDK or Runtime Environment.
3. Log on again or run the updated shell script to activate the new **PATH** setting.
4. If you installed the Java Plug-in, remove the Java Plug-in files from the web browser directory.

Chapter 3. Using the ahead-of-time compiler and WebSphere Real Time

The Real-Time JVM works like a standard IBM JVM. You can run it with the JIT enabled using the **-Xjit** option or without the JIT, but with precompiled code, using the **-Xnojit** option on the command line. The latter can be used only when **-Xrealtime** is specified.

WebSphere Real Time introduces the ability to compile Java files before execution, thus removing one of the causes of the variations of time that an application might take. Ahead-of time (AOT) compilation replaces just-in-time (JIT) compilation when you run your application with the **-Xrealtime** and the **-Xnojit** options.

Attention: You cannot use the **-Xdebug** option and the **-Xnojit** option with code that has been precompiled using the ahead-of-time compiler because **-Xdebug** compiles code in a different way from the AOT compiler and is not supported.

To debug your code, use interpreted or JIT-compiled code.

The ahead-of-time compiler

Ahead-of-time (AOT) compilation allows you to compile Java classes before you execute your code. AOT compilation avoids the problem of the JIT compiler not compiling code before it is required for execution on a sensitive performance path. You can perform AOT compilation on your code, and thus reduce the complexity of ensuring that your code is compiled at runtime through some form of warm-up.

Note: AOT-generated code does not perform as well as JIT-generated code, although it usually performs better than interpreted code.

Just-in-time compilation does not cause non-deterministic delays in real-time code. The JIT compiler runs as a high-priority SCHED_OTHER thread, running above the priority of standard Java threads, but running below the priority of real-time threads. High priority work is not preempted by the JIT compiler, as a result, important real-time work is done on time. However, real-time code could end up running interpreted code because the JIT has not had time to compile the hot methods that have queued up. A comparison of AOT compilation and JIT compilation is shown in Figure 2 on page 16.

In general, if your application has a warm-up phase, it is more efficient to run with the JIT and, if necessary, disable the JIT when the warm-up phase is complete. This approach allows the JIT compiler generate code for the environment where your application runs.

If the application has no warm-up phase and it is not clear if key paths of execution are compiled through standard application operation, AOT compilation works well in this environment.

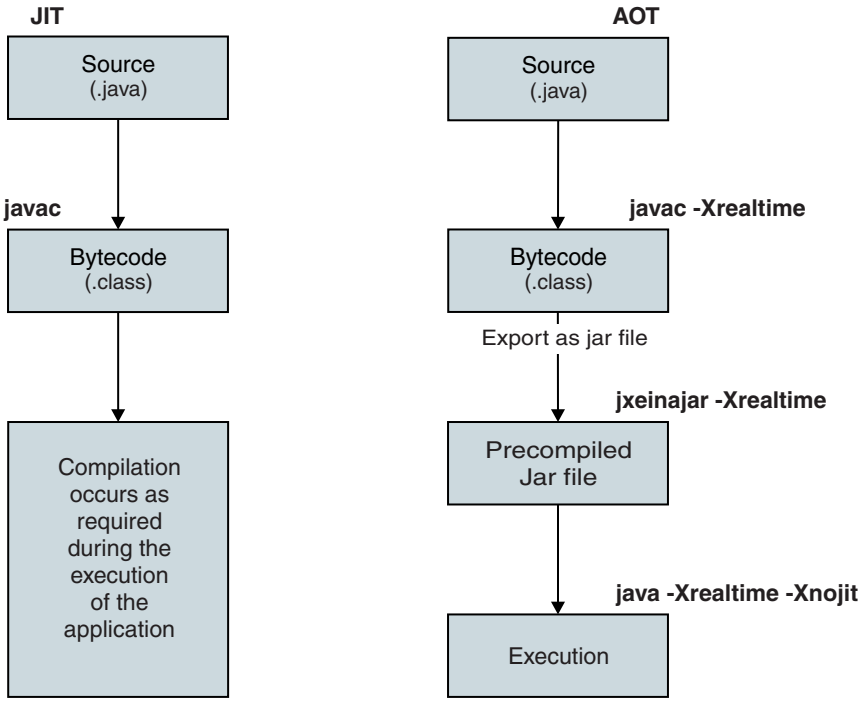

Figure 2. Comparing JIT compiler and AOT compiler.

Using the AOT compiler

Use these steps to precompile your Java code. This procedure describes the use of **-Xrealtime** in a javac command, the jxeinajar tool, and **-Xrealtime** and **-Xnojit** options with the java command.

Using the ahead-of-time compiler means that compilation is separate from the execution time of the application. Also, you can compile more methods at once rather than just the frequently used methods. You can compile everything in an application or just individual classes, as shown in the following steps:

1. From a shell prompt, enter:

   ```
   javac -Xrealtime source
   ```

   This command creates the Java bytecode from your source for use in the real-time environment. See Figure 2.
2. Package the class files generated into a jar file. For example: to create test.jar:

   ```
   jar cvf test.jar source
   ```
3. From a shell prompt, enter:

   ```
   jxeinajar -Xrealtime -outPath ./aot test.jar
   ```

   This command precompiles the *test.jar* file and writes the output to the output directory ./aot.
4. From a shell prompt, enter:

   ```
   java -Xrealtime -Xnojit ./aot/test.jar
   ```

   This command runs the precompiled file.

Building precompiled files

You can build all, some, or include IBM-provided Java classes into a precompiled jar file. This process uses the **-Xrealtime** option with javac and the jxeinajar tool to create the precompiled file.

Precompilation of jar files with jxeinajar is supported only when subsequently running with the **-Xrealtime** option. If you need to use the same jar files for both real-time and standard execution, do **not** precompile them.

You can precompile jar files using the jxeinajar tool. jxeinajar enables you to build your application in one of three ways.

Note:

- If you have set a timeout on your Linux system, you might need to override it when precompiling large jar files; otherwise, compilation will timeout and the jar file is not created.
- If you have digitally signed your jar file using jarsigner, you must re-sign the jar file after performing jxeinajar processing since the jxeinajar processing alters the contents of the jar file.

Precompiling all classes and methods in an application

This procedure precompiles all the classes in an application. It builds a set of optimized jar files from a corresponding set of ordinary jar files. The optimized jar files have all methods compiled.

For the purposes of this example, the application resides under the directory specified by the environment variable *$APP_HOME* and the jar files are in the subdirectory **$APP_HOME**/lib. The application also uses some classes from the IBM-provided classes in core.jar. In this case, you can just precompile the application code, namely main.jar and util.jar.

1. From a shell prompt, enter `cd` **`$APP_HOME`**.

   where *$APP_HOME* is the directory of your application.
2. From a shell prompt, enter `mkdir` *`aot`* to create a new directory called *aot* for the output.
3. From a shell prompt, enter `cd` **`$APP_HOME`**`/lib`. **$APP_HOME**/lib is the directory where main.jar and util.jar are stored.
4. From a shell prompt, enter `jxeinajar -Xrealtime -outPath` **`$APP_HOME`**`/aot` This procedure optimizes each of the jar files found in **$APP_HOME**/lib, writing out progress information to the screen, and then creating the new jar file in the **$APP_HOME**/aot directory.

For more options, specify: `jxeinajar -help -Xrealtime`.

Precompiling frequently used methods

You can use profile-directed AOT compilation to precompile only the methods that are frequently used by the application. It builds a set of optimized jar files from a corresponding set of ordinary jar files and a set of option files previously generated by running the application with a special option: **-Xjit:verbose={precompile},vlog=***optFile*. The generated options file contains a list of methods that will be compiled.

Before you start, create a list of those methods that would be compiled by a JIT compiler.

Note: You can edit the file generated by the **-Xjit:verbose={precompile}** option. It is just an explicit specification of the methods that are to be precompiled. These methods are specific, that is, they contain the full signature for each method to be compiled, which lets you compile `com/acme/foozle.myMethod(J)V` but not `com/acme/foozle.myMethod(I)V`.

1. From a shell prompt, enter `cd` *$APP_HOME* where *$APP_HOME* is the directory of your application.
2. From a shell prompt, enter

```
java -Xjit:verbose={precompile},vlog=$APP_HOME/app.precompileOpts -cp $APP_HOME/lib/demo.jar applicationName
```

   where:

   - *app.precompileOpts* is the name of the log file that will list the methods compiled with JIT
   - *applicationName* is the name of your application.

   This command creates a list of the methods that are compiled using JIT.
3. From a shell prompt, enter `cd` *$APP_HOME*`/lib` *$APP_HOME/lib* is the directory where your application's jar files are stored.
4. From a shell prompt, enter:

```
jxeinajar -Xrealtime -outPath $APP_HOME/aot -optFile $APP_HOME/app.precompileOpts
```

   where:

   - *app.precompileOpts* is the name of the log file that lists the methods compiled with JIT that you want to have precompiled,
   - The resulting jar file is stored in *$APP_HOME*`/aot`.
5. To compile realtime.jar, vm.jar,

```
jxeinajar -Xrealtime -searchPath $JAVA_HOME/jre/bin/realtime/jclSC150/ -outPath $APPHOME/aot
                                                                    -optFile sample.precompileOpts
```

6. Compile core.jar

```
jxeinajar -Xrealtime $JAVA_HOME/jre/lib/core.jar -outPath $APPHOME/aot -optFile sample.precompileOpts
```

7. Compile all of the sample application methods

```
jxeinajar -Xrealtime -outPath $APPHOME/aot demo.jar
```

   This command compiles all of the sample application methods.
8. To test this command run your application with the **-nojit** option, which uses the code in the precompiled jars. From the shell prompt, enter:

```
java -Xrealtime -Xnojit -Xbootclasspath/p:$APPHOME/aot/core.jar:$APPHOME/aot/vm.jar:$APPHOME/aot/realtime.jar
                                                  -cp $APPHOME/aot/demo.jar applicationName
```

   where *applicationName* is the name of your application.

Precompiling files provided by IBM

You can precompile files provided by IBM, for example core.jar, to achieve a compromise between performance and predictability.

The precompilation is similar to the task of precompiling your application jars but there is an additional requirement at runtime; you must ensure that your boot classpath is specified correctly to use these files instead of the files in the JRE. You can do this using the **-Xbootclasspath/p** option, which instructs the JVM to look first in the specified boot classpath location ahead of the default location.

Precompile core.jar for use with the application:

1. From a shell prompt, enter `cd` **$JAVA_HOME**`/lib`. Where **$JAVA_HOME** is your Java home directory.

2. To run the jxeinajar tool, at a shell prompt, enter

   ```
   jxeinajar -Xrealtime -outPath $APP_HOME/aot core.jar
   ```

   This command precompiles the IBM-provided file called core.jar.
3. Run your application specifying the **-Xbootclasspath/p** option to use the precompiled jar file. To run your application, enter:

   ```
   java -Xrealtime -Xnojit -Xbootclasspath/p:$APP_HOME/aot/core.jar
           -classpath:$APP_HOME/aot/main.jar:$APP_HOME/aot/util.jar ...
   ```

The Just-In-Time (JIT) compiler

You can control when and how the JIT compiler operates using the java.lang.Compiler class that is provided as part of the standard SDK class library. IBM fully supports the Compile.compileClass(), Compiler.enable() and Compiler.disable() methods.

For example, if you want to "warm-up" your application and know that the key methods in your application have been compiled, you can call the Compiler.disable() method after you have warmed up your application and be confident that JIT compilation will not occur during the remainder of the execution of your application.

You can control method compilation in two ways:

- Specify a set of methods that you can compile:

  ```
  Compiler.command("{<method specification>}(compile)");
  ```

  where <*method specification*> is a list of all the methods that have been loaded at this point and should be compiled. <*method specification*> describes a fully qualified method name. An asterisk denotes a wild-card match.

  For example, if you wanted to compile all methods starting with java.lang.String that were already loaded, you would specify:

  ```
  Compiler.command("{java.lang.String*}(compile)");
  ```

  Note: This command compiles not only methods in the java.lang.String class, but also in the java.lang.StringBuffer class, which might not be what you wanted. If you wanted to compile only methods in the java.lang.String class, you specify:

  ```
  Compiler.command("{java.lang.String.*}(compile)");
  ```

- Specify that all methods in the compilation queue should be compiled before execution of this thread continues:

  ```
  Compiler.command("waitOnCompilationQueue");
  ```

  You might want to ensure that the compilation queue was empty before disabling the compiler. A typical technique for compiling a set of methods and classes could be:

```
Compiler.enable();                              // ensure compiler is active
Compiler.command("{com.mycompany.*}(compile)"); // queue up all the methods you want to compile
Compiler.command("waitOnCompilationQueue");     // wait until all those methods are compiled
Compiler.disable();                             // turn the compiler off
```

Determinism during JNI transitions

By default, the JIT will generate optimized code for high performance Java-to-native JNI transitions. There is a small possibility of reduced determinism when reloading a native library using the following code sequence:

```
RegisterNatives / UnregisterNatives / RegisterNatives
```

To revert to the slower, more deterministic code, use the command line option **-Xjit:disableDirectToJNI**.

Enabling the JIT

The JIT can be explicitly enabled in a number of different ways. Both command-line options override the **JAVA_COMPILER** environment variable.

- Set the **JAVA_COMPILER** environment variable to "jitc" before running the Java application. At a shell prompt, enter:
  - **For the Korn shell:** `export JAVA_COMPILER=jitc`

    Tip: (Korn shell commands are used for the remainder of this User Guide.)
  - **For the Bourne shell:**

    ```
    JAVA_COMPILER=jitc
    export JAVA_COMPILER
    ```
  - **For the C shell:** `setenv JAVA_COMPILER jitc`

  If the **JAVA_COMPILER** environment variable is an empty string, the JIT remains disabled. To disable the environment variable, at the shell prompt, enter `unset` **`JAVA_COMPILER`**.
- Use the **-D** option on the JVM command line to set the java.compiler property to "jitc". At a shell prompt, enter: `java -Djava.compiler=jitc <MyApp>`
- Use the **-Xjit** option on the JVM command line. You must **not** specify the **-Xint** option at the same time. At a shell prompt, enter: `java -Xjit <MyApp>`

Disabling the JIT

The JIT can be disabled in a number of different ways. Both command-line options override the **JAVA_COMPILER** environment variable.

- Set the **JAVA_COMPILER** environment variable to "NONE" or the empty string before running the Java application. Type the following at a shell prompt:
  - **For the Korn shell:** `export JAVA_COMPILER=NONE`

    Tip: (Korn shell commands are used for the remainder of this information.)
  - **For the Bourne shell:**

    ```
    JAVA_COMPILER=NONE
    export JAVA_COMPILER
    ```
  - **For the C shell:** `setenv JAVA_COMPILER NONE`
- Use the **-D** option on the JVM command line to set the java.compiler property to "NONE" or the empty string. Type the following at a shell prompt: `java -Djava.compiler=NONE <MyApp>`
- Use the **-Xint** option on the JVM command line. Type the following at a shell prompt: `java -Xint <MyApp>`

Determining whether the JIT is enabled

You can determine the status of the JIT using the **-version** option.

Enter the following at a shell prompt: `java -version`

If the JIT is not in use, a message is displayed that includes the following: `(JIT disabled)`

If the JIT is in use, a message is displayed that includes the following: `(JIT enabled)`

Chapter 4. Using the Metronome Garbage Collector

Metronome Garbage Collector replaces the standard Garbage Collector in WebSphere Real Time.

Introduction to the Metronome Garbage Collector

The benefit of Metronome is that the time it takes is more predictable and garbage collection can take place at set intervals over a period of time.

The key difference between Metronome garbage collection and standard garbage collection is that Metronome garbage collection occurs in small interruptible steps but standard garbage collection stops the application while it marks and collects garbage.

You can control garbage collection with the Metronome Garbage Collector using the **-Xgc:targetUtilization=N** option to limit the amount of CPU used by the Garbage Collector.

For example:

```
java -Xrealtime -Xgc:targetUtilization=80 yourApplication
```

The example specifies that your application will run for 80% in every 10ms. The remaining 20% of the time is used for garbage collection. The Metronome Garbage Collector guarantees utilization levels provided that it has been given sufficient resources. Garbage collection will begin when the heap reaches half full.

Garbage collection and priorities

The garbage collection thread has to run at a priority higher than the highest priority thread that generates garbage in the heap; otherwise, it might not run as specified by the configured utilization. Both regular Java threads and real-time threads can generate garbage and, therefore, garbage collection must run at a priority higher than all regular and real-time threads. This is handled automatically by the JVM and garbage collection runs at 0.5 priority above the highest priority of all regular and real-time threads. However, it is important to note that, to ensure that no-heap real-time threads (NHRTs) are not affected by garbage collection, all NHRTs run at a priority that is higher than the highest priority real-time threads. This prioritization ensures that the NHRTs run at a priority higher than garbage collection and are not delayed.

Table 7 shows a typical example of priorities that you can define and the related garbage collection priorities that follow from your choice

For the comparison of Java priorities and OS priorities, see “Priority mapping” on page 31.

Table 7. Example of garbage collection and priorities

Threads	**Priorities (examples)**
If the highest priority real-time thread is:	20 (OS priority 43)
Then the Garbage Collector is:	20.5 (OS priority 44)

Table 7. Example of garbage collection and priorities (continued)

Threads	Priorities (examples)
To ensure that a NHRT runs independently of the garbage collector, set a higher priority than the GC:	21 (OS priority 45) or higher.
The Metronome alarm thread is:	Priority 46 (OS priority 89)

Note: Even with this configuration, no-heap real-time threads are not completely unaffected by garbage collection because the metronome alarm thread runs at the highest priority in the system to ensure that it can wake up regularly and work out if garbage collection needs to do anything. The work to do that is, of course, tiny and thus not a major consideration.

Metronome and class unloading

Metronome does not unload classes when they are no longer referenced. Typically, you preload classes that your application references to guarantee that class loading does not occur in a time-critical section of code. Class preloading is a necessary precaution because loading an unreferenced class from disk can take in the order of milliseconds and verifying a class can take in the order of hundreds of microseconds (depending on class complexity). Because classes are not unloaded and because each class requires a small amount of immortal memory, an application that references a significant number of classes might need to increase the size of the immortal memory area. These factors also apply for both anonymous and generated classes.

Metronome threads

The Metronome Garbage Collector consists of two types of threads: an alarm thread and a number of collection threads.

The Metronome Garbage Collector periodically checks the JVM to see if there is sufficient free space in heap memory. When the amount of free space falls below the limit, it triggers the JVM to start garbage collection.

Alarm thread

There is one alarm thread and it is guaranteed to use minimal resources. It "wakes" at regular intervals and checks:

- The amount of free space in the heap memory
- Whether garbage collection is currently taking place

If there is insufficient free space available and no garbage collection is taking place, the alarm thread triggers the collection threads to start garbage collection. The alarm thread does nothing until the next scheduled time for it to check the JVM.

Collection threads

Each collection thread checks Java and real-time threads for heap objects. They check the memory areas in the following sequence:

1. Scoped memory to identify and mark any live objects in the heap that are being used by objects from scoped memory.
2. Immortal memory to identify and mark any live objects in the heap that are being used by objects from immortal memory.
3. Heap memory to identify and mark live objects.

When the live objects have been marked, the unmarked objects are available for collection.

Metronome can run multiple garbage collection threads. The number of GC threads is set equal to or less than the number of physical processors. You set the number of threads for your JVM using the **-Xgc:threads** option.

Troubleshooting the Metronome Garbage Collector

Using the **-verbose:gc** option, you can control the frequency of metronome garbage collection, out of memory exceptions, and the metronome behavior on explicit system calls.

Using the -verbose:gc option

You can use the **-verbose:gc** option with the **-Xgc:verboseGCCycleTime=N** option to write information to the console about Metronome GC activity. Not all XML properties in the **-verbose:gc** output from the standard JVM appear or apply to the output of Metronome Garbage Collector.

The **-verbose:gc** option allows you to view the minimum, maximum, and mean free space in the heap and the immortal memory. In this way you can check the level of activity and use in these memory areas and subsequently adjust the values if necessary. The **-verbose:gc** option in real time writes metronome statistics to the console.

The **-Xgc:verboseGCCycleTime=N** option controls the frequency of retrieval of the information. It determines the time in milliseconds that the summaries should be dumped. The default value for N is 1000 milliseconds. The cycle time does not mean the summary is dumped precisely at that time, but when the last GC cycle that meets this time criteria passes. The collection and display of these statistics can distort the metronome and, as N gets smaller, the distortion can get quite large.

A quantum describes a single episode of Metronome Garbage Collector activity.

Example

Enter:

```
java -Xrealtime -verbose:gc -Xgc:verboseGCCycleTime=N myApplication
```

The verbose garbage collection output looks like this:

```
<gc type="heartbeat" id="5" timestamp="Mon Dec 18 09:44:14 2006" intervalms="1523.189">
  <summary quantumcount="211">
    <quantum minms="0.016" meanms="0.467" maxms="0.654" />
    <heap minfree="137461760" meanfree="180471235" maxfree="273989632" />
    <immortal minfree="15949248" meanfree="15949248" maxfree="15949248" />
  </summary>
</gc>
```

Where the tags in this example:

quantumcount
Is the number of GC quanta run in the summary period.

quantum
Provides information on all quanta in the summary period.

minms
Is the minimum Metronome Garbage Collector quanta pause time in milliseconds for the given cycle.

maxms
Is the maximum Metronome Garbage Collector quanta pause time in milliseconds for the given cycle.

meanms
Is the mean Metronome Garbage Collector quanta pause time in milliseconds for the given cycle.

immortal
Is the immortal memory.

minfree
Is the minimum amount of free memory in the Java heap and immortal heap for the given interval, sampled at the end of each GC quantum.

meanfree
Is the mean amount of free memory in the Java heap and immortal heap for the given interval, sampled at the end of each GC quantum.

maxfree
Is the maximum amount of free memory in the Java heap and immortal heap for the given interval, sampled at the end of each GC quantum.

type The values are:

heartbeat

synchgc
See "Synchronous garbage collections" on page 25

priogc priority related information, see "Priority changes" on page 25

See also "Priority changes" on page 25.

intervalms
Is the actual time, in milliseconds, since the last summary was written, corresponding to the **-Xgc:verboseGCCycleTime=N** option. When the type=heartbeat is used, this is the frequency of the sampling.

id Is the number of this sample.

timestamp
Is the date and time of this sample.

Note:

- If only one GC quantum occurred in the interval between two heartbeats, the free memory is sampled only at the end of this one quantum, and therefore the minimum, maximum and mean amounts given in the heartbeat summary are all equal.
- It is possible that the interval might be significantly larger than the cycle time specified because the GC has no work on a heap that is not full enough to warrant GC activity. For example, if your program requires GC activity only once every few seconds, you are likely to see a heartbeat only once every few seconds as well.

  Also, if an event such as a synchronous GC or a priority change occurs, then we do not wait until the full interval (N milliseconds) has elapsed, but output the details of the event immediately.
- If the maximum GC quanta for a given period is too large, you might want to reduce the target utilization (**-Xgc:targetUtilization** option) to give the GC more time to do work, or you might want to increase the heap

size (**-Xmx** option). Similarly, if your application can tolerate longer delays than are currently being reported, you can increase the target utilization or decrease the heap size.

- The output can be redirected to a log file instead of the console using the **-Xverbosegclog:**<*file*> option; for example, **-Xverbosegclog:out** writes the verbose gc output to the file *out*.

Priority changes

In addition to summaries, an entry is written to the verbose GC log when the GC thread priority changes (as a result of the application changing thread priorities, or one or more threads in an application terminating). An example of a GC thread priority change entry is:

```
<gc type="priogc" id="64" timestamp="Thu May 25 13:16:25 2006" intervalms="0.631">
   <priority newpriority="84" />
</gc>
```

Note: The priority listed is the underlying OS thread priority, not a Java thread priority.

Synchronous garbage collections

An entry is also written to the verbose GC log when a synchronous (non-deterministic) GC occurs. This event has two possible causes: an explicit System.gc() call in the code, or the JVM running out of memory and performing a synchronous GC to avoid an OutOfMemoryError condition.

An entry is also written to the verbose GC log when an explicit System.gc() call is made. An example of a System.gc() entry is:

```
<gc type="synchgc" id="84" timestamp="Tue Jun 06 13:58:33 2006" intervalms="991.494">
     <details reason="system garbage collect" />
     <heap freebytes="488521728" />
     <immortal freebytes="93561320" />
</gc>
```

An example of a synchronous GC entry as a result of an out of memory condition is:

```
<gc type="synchgc" id="63" timestamp="Tue Jul 04 04:34:19 2006" intervalms="2136.954">
     <details reason="out of memory" />
     <heap freebytes="0" />
     <immortal freebytes="16250596" />
</gc>
```

Out of memory entries

When one of the memory areas runs out of free space, an entry is written to the verbose GC log before the OutOfMemoryError is thrown. An example of this output is:

```
<event details="out of memory" timestamp="Thu Aug 03 10:49:18 2006" memoryspace="Scoped" J9MemorySpace="0x08482F30" />
```

By default (unless you have overridden the settings with **-Xdump:none**) a javadump is produced as a result of an OutOfMemoryError. This contains information about the memory areas used by your program. Together with the J9MemorySpace value given in the verbose GC output, this can be used to identify the particular memory area that ran out of space:

```
0SECTION       MEMINFO subcomponent dump routine
NULL           ================================
<< output removed for clarity >>
1STSEGTYPE     Object Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGSUBTYPE  Scoped Segment  ID=08482F30
1STSEGMENT     08482780 08807C48 08816748 08816748  00002008 eb00
1STSEGSUBTYPE  Immortal Segment  ID=08482F14
1STSEGMENT     08482708 B26EA008 B36EA008 B36EA008  00001008 1000000
1STSEGSUBTYPE  Heap Segment  ID=08482EF8
1STSEGMENT     08482690 B36EB008 B76EB008 B76EB008  00000009 4000000
NULL
```

In the example above, the memory space ID given in the verbose GC output (0x08482F30) can be matched to the ID of the ″Scoped Segment″ memory area in the javadump. This can be useful if you have several scopes and need to identify which one has gone out of memory, since the verbose GC output will only indicate whether the OutOfMemoryError occurred in immortal, scoped or heap memory, rather than down to the level of individual scopes.

Metronome behavior in out of memory conditions

By default, metronome triggers an unlimited, non-deterministic garbage collection when the JVM runs out of memory. To prevent non-deterministic behavior, use the **-Xgc:noSynchronousGCOnOOM** option to throw an OutOfMemoryError when the JVM runs out of memory.

The default unlimited collection will run until all possible garbage is collected in a single operation. The pause time required will usually be many milliseconds greater than a normal metronome incremental quantum.

Related information

Using -Xverbose:gc to analyze synchronous garbage collections

Metronome behavior on explicit System.gc() calls

By default, Metronome performs a full synchronous GC when a System.gc() call is issued from a Java application. Use this to clean up the heap in a controlled manner. It is a non-deterministic operation because it performs a complete GC before returning.

Some applications call third party software that has `System.gc()` calls where it is not acceptable to create these non-deterministic delays. If an application wants to have `System.gc()` calls return immediately instead of performing a full GC, you can specify the **-Xdisableexplicitgc** option.

Note: Using the **-Xrealtime** option with **-verbose:gc** generates this output:

```
java -Xrealtime -verbose:gc -Xgc:verboseGCCycleTime=N myApplication
```

The verbose garbage collection output looks like this:

```
<gc type="heartbeat" id="5" timestamp="Mon Dec 18 09:44:14 2006" intervalms="1523.189">
  <summary quantumcount="211">
    <quantum minms="0.016" meanms="0.467" maxms="0.654" />
    <heap minfree="137461760" meanfree="180471235" maxfree="273989632" />
    <immortal minfree="15949248" meanfree="15949248" maxfree="15949248" />
  </summary>
</gc>
```

Metronome Garbage Collector limitations

There might be pauses caused by garbage collection when using the Metronome Garbage Collector.

One limitation of the system:

- Root scanning: the ″roots″ of the garbage collection are variables on the stack, in global variables, and objects in immortal and scoped memory. Each active thread stack is processed in an uninterruptible step. This means that if you have very deep stacks, the performance of the system might be worse than expected with extended garbage collection pauses at the beginning of collection. Furthermore, although the immortal memory is processed incrementally, all other scoped memory areas are currently processed in one atomic, uninterruptible step. This means that if you make significant use of scoped memory, the performance of the system might be worse than expected with extended garbage collection pauses during the phase of root scan when it is processing scoped memory.

Tuning Metronome Garbage Collector

You can tune the real-time environment by controlling the amount of memory that your application uses. For example, use the **-Xmx**, **-Xgc:immortalMemorySize=size**, **-Xgc:scopedMemoryMaximumSize=size**, and the **-Xgc:targetUtilization=N** options.

- Use the **-Xmx** option to limit the size of the heap.

  The value chosen is used as the upper limit of heap size and thus should reflect the likely usage over time. Choosing a value that is too low increases the garbage collection frequency and leads to a lower overall throughput although it reduces the memory footprint. For good real-time performance, avoid paging. It is normal to ensure that the footprint of all the active processes on a machine does not exceed the physical memory size.
- Use the **-Xgc:immortalMemorySize=size** option to control the size of the immortal memory area.

  You must analyze carefully the use of immortal memory. The “ideal” application uses immortal memory during startup but thereafter stops using it. If allocation of immortal objects continues, the application is able to continue to run until immortal memory has been exhausted. The current usage can be obtained by adding:

  ```
  long used = ImmortalMemory.instance().memoryConsumed();
  ```

  to your code.
- Use the **-Xgc:scopedMemoryMaximumSize=size** option to ensure that applications do not request excessive amounts of scoped memory. You would use this option for diagnosis rather than tuning.
- Set the **-Xgc:targetUtilization=N** option to ensure that under the worst case conditions (maximum allocation rate of heap objects), the garbage collector can collect garbage at a higher rate than the application generates it.

  Typically the default value should be sufficient but application performance might be improved by increasing the utilization to the point at which the collector is able to collect garbage slightly faster than the application can create it.

Chapter 5. Support for RTSJ

WebSphere Real Time implements the Real-Time Specification for Java (RTSJ).

WebSphere Real Time version 1.0 has been certified as RTSJ 1.0.1b compliant against the RTSJ Technology Compatibility Kit 1.0.1 and 1.0.2 version J9 3.0.10 FCS2 and is compliant with the Java Compatibility Kit (JCK) at version 5.0.

Thread scheduling and dispatching

The Linux operating system supports three scheduling policies, one for standard threads (SCHED_OTHER) and two for real-time applications: SCHED_FIFO and SCHED_RR. Only SCHED_OTHER and SCHED_FIFO are used by WebSphere Real Time.

The kernel decides which is the next runnable thread to be executed by the CPU. The kernel maintains a list of runnable threads. It looks for the thread with the highest priority and selects that thread as the next thread to be run.

Thread priorities can be listed using the following command:

```
ps -emo pid,ppid,tid,comm,rtprio,cputime
```

The output looks like this:

```
  PID    PPID    TID   COMMAND    RTPRIO     TIME
18753   18752      -      java         -     00:00:02
    -       -   18753         -         -     00:00:00
    -       -   18756         -        89     00:00:00
    -       -   18762         -        12     00:00:00
    -       -   18763         -        11     00:00:00
    -       -   18764         -        89     00:00:00
    -       -   18765         -        43     00:00:00
    -       -   18766         -        83     00:00:00
```

This output shows the Java process, the main thread with priority "-" (other), and some real-time threads with priorities from 11 to 89.

SCHED_OTHER
The default universal time-sharing scheduler policy that is used by most threads. These threads must be assigned with a priority of zero.

SCHED_OTHER uses time slicing, which means that each thread runs for a limited time period, after which the next thread is allowed to run.

SCHED_FIFO
Can be used only with priorities greater than zero. This usage means that when a SCHED_FIFO process becomes available it preempts any normal SCHED_OTHER thread.

If a SCHED_FIFO process that has a higher priority becomes available, it preempts an existing SCHED_FIFO process if that process has a lower priority. This thread is then kept at the top of the queue for its priority.

There is no time slicing.

SCHED_RR
Is an enhancement of SCHED_FIFO. The difference is that each thread is

allowed only to run for a limited time period. If the thread exceeds that time, it is returned to the list for its priority.

Note: SCHED_RR is not used by WebSphere Real Time.

For details on these Linux scheduling policies, see the man page for **sched_setscheduler**. To query the current scheduling policy, use **sched_getscheduler**.

Relationships of the real-time threads

There are two main types of realtime schedulables: real-time threads and asynchronous event handlers.

These schedulables have the following parameters associated with them:

SchedulingParameters
PriorityParameters schedules real-time schedulables by priority.

ReleaseParameters

- PeriodicParameters describes periodic release of realtime schedulables. A periodic real-time thread is one that is released at regular intervals.
- AperiodicParameters describes the release of realtime schedulables. Aperiodic real-time threads are released at irregular intervals.

MemoryParameters
Describes memory allocation constraints for realtime schedulables.

ProcessingGroupParameters
Unsupported in WebSphere Real Time.

The priority scheduler

In WebSphere Real Time, the scheduler is a priority scheduler. As its name implies, it orders the execution of the schedulable objects according to the active priority.

Each realtime schedulable has an associated priority. The scheduler maintains the list of schedulable objects and determines when each object can be released for execution within the CPU. The scheduler must abide by the various parameters that are associated with each schedulable object. The methods addToFeasibility, isFeasible, and removeFromFeasibility are provided for this purpose.

Priorities and policies

Regular Java threads, that is, threads allocated as `java.lang.Thread` objects, use the default scheduling policy of SCHED_OTHER. Real-time threads, that is, threads allocated as `java.lang.RealtimeThread`, and asynchronous event handlers use the SCHED_FIFO scheduling policy.

Regular Java threads have the operating system thread priority set to 0, even though you might have programmatically specified a Java thread priority from 1 to 10. For real-time threads, the SCHED_FIFO policy has no time slicing and supports 99 priorities from 1 (the lowest) to 99 (the highest). This WebSphere Real Time implementation supports 28 user priorities in the range 11-38 inclusive, so:

```
javax.realtime.PriorityScheduler().getMinPriority()
```

returns 11 and,

```
javax.realtime.PriorityScheduler().getMaxPriority()
```

returns 38.

OS priorities 81 to 89 are used by the IBM JVM for dispatching worker threads. These threads are all designed to do a very small amount of work before going back to sleep. The threads are as follows:

- The Metronome Garbage Collector alarm thread runs at a priority of 89. It wakes up regularly and dispatches a GC work unit.
- Two asynchronous signal threads, which process asynchronous signals, one being a no-heap real-time thread at priority 88 and the other at priority 87.
- Two Timer threads, which dispatch timer events, one being a no-heap real-time thread for no-heap timers at priority 85 and the other at priority 83.
- The Async Event Handler Threads, which are dispatched to run asynchronous event handlers, and while running an async event handler are assigned the priority of that handler. The system starts up with two no-heap real-time handler threads at priority 85 and 8 others at priority 83.
- The asynchronous signal no-heap realtime thread at priority 88 handles requests for heap dumps, core dumps and javacore dumps. It temporarily boosts its priority to 89 while creating dump files.

The Metronome GC Trace thread runs at OS priority 12, and the JIT Sampler thread (which samples java methods for compilation) runs at OS priority 13.

The JIT Compilation thread (which is different from the JIT Sampler thread) runs with the SCHED_OTHER policy at OS priority 0.

Note: Both the JIT compilation and JIT sampler threads are not active if **-Xnojit** or **-Xint** is specified.

The Metronome Garbage Collector and finalizer priority constantly changes (prior to each round of collection) to be above the highest priority heap allocating thread. You must ensure that the priority of heap allocating threads is below that of NoHeapRealtimeThreads.

A heap allocating thread is any non-NHRT user thread that is not asleep or blocked on a monitor. A user thread executing native code outside the JNI interface is not considered to be heap-allocating. If a garbage collection is in progress when a heap allocating thread wakes up, is no longer blocked on a monitor, or leaves JNI, then it will be forced to wait until the garbage collection has finished before it can continue.

OS priority 81 is reserved for internal JVM threads that are allocating from the heap. If there exists an internal JVM thread at OS priority 81, the garbage collector runs at OS priority 82. When the only heap-allocating user threads are not real-time threads, the GC priority runs at OS priority 11. Otherwise, the GC runs at a priority that is one OS priority higher than the highest priority heap-allocating user thread.

The GC priority is adjusted just before a round of collection.

Priority mapping

Each Java priority is mapped to an associated operating system base priority, and each operating system priority is associated with a scheduling policy. The WebSphere Real Time Linux operating system scheduling policies are SCHED_OTHER and SCHED_FIFO.

The following table shows how the Java priorities are mapped to native operating system priorities. Some Java priorities are reserved for use by the JVM, while some native priorities that have no corresponding Java priorities are used by the JVM as well.

Note:

- Priorities 1-10 are used by standard Java Threads. Priorities 11 upwards are used by real-time threads and no-heap real-time threads.
- A schedulable object always runs with its active priority. The active priority is initially the schedulable object's base priority, but the active priority can be temporarily raised by priority inheritance. The base priority of a schedulable can be changed while the schedulable is running.

User base priorities:

```
Java priorities 1-10: SCHED_OTHER, OS priority 0

Java priority 11: SCHED_FIFO, OS priority 25
Java priority 12: SCHED_FIFO, OS priority 27
Java priority 13: SCHED_FIFO, OS priority 29
Java priority 14: SCHED_FIFO, OS priority 31
Java priority 15: SCHED_FIFO, OS priority 33
Java priority 16: SCHED_FIFO, OS priority 35
Java priority 17: SCHED_FIFO, OS priority 37
Java priority 18: SCHED_FIFO, OS priority 39
Java priority 19: SCHED_FIFO, OS priority 41
Java priority 20: SCHED_FIFO, OS priority 43
Java priority 21: SCHED_FIFO, OS priority 45
Java priority 22: SCHED_FIFO, OS priority 47
Java priority 23: SCHED_FIFO, OS priority 49
Java priority 24: SCHED_FIFO, OS priority 51
Java priority 25: SCHED_FIFO, OS priority 53
Java priority 26: SCHED_FIFO, OS priority 55
Java priority 27: SCHED_FIFO, OS priority 57
Java priority 28: SCHED_FIFO, OS priority 59
Java priority 29: SCHED_FIFO, OS priority 61
Java priority 30: SCHED_FIFO, OS priority 63
Java priority 31: SCHED_FIFO, OS priority 65
Java priority 32: SCHED_FIFO, OS priority 67
Java priority 33: SCHED_FIFO, OS priority 69
Java priority 34: SCHED_FIFO, OS priority 71
Java priority 35: SCHED_FIFO, OS priority 73
Java priority 36: SCHED_FIFO, OS priority 75
Java priority 37: SCHED_FIFO, OS priority 77
Java priority 38: SCHED_FIFO, OS priority 79
```

Internal base priorities:

```
Internal Java priority 39: SCHED_FIFO, OS priority 81
Internal Java priority 40: SCHED_FIFO, OS priority 83
Internal Java priority 41: SCHED_FIFO, OS priority 84
Internal Java priority 42: SCHED_FIFO, OS priority 85
Internal Java priority 43: SCHED_FIFO, OS priority 86
Internal Java priority 44: SCHED_FIFO, OS priority 87
Internal Java priority 45: SCHED_FIFO, OS priority 88
OS priorities 11, 12, 13
OS priorities even numbers 26, 28, 30, ..., 82
OS priority 89
```

See also: the ″Synchronization″ section in the http://www.rtsj.org/specjavadoc101b/book_index.html.

Priority inheritance

The active priority of a thread can be temporarily boosted because it holds a lock required by a higher priority thread. These could be internal JVM locks or user

level monitors associated with synchronized methods or synchronized blocks. The priority of a regular Java thread, therefore, could temporarily have a real-time priority until the point at which the thread has released the lock.

For more information about base and active priorities, see the ″Synchronization″ section in the RTSJ specification.

Note: One consequence of priority inheritance means that the thread policy of a SCHED_OTHER thread is temporarily changed to SCHED_FIFO.

Memory management

Garbage-collected memory (heaps) have always been considered an obstacle to real-time programming because of the unpredictable behavior introduced by the Garbage Collector. The Real-Time Specification for Java (RTSJ) addresses this problem by providing several extensions to the memory model outside the garbage-collected heap for both short-lived and long-lived objects.

Memory areas

The RTSJ introduces the concept of a memory area that may be used for the allocation of objects. Some memory areas exist outside the heap and place restrictions on what the system and garbage collector may do with objects. For example, objects in some memory areas are never garbage collected but the Garbage Collector can scan these memory areas for references to any object within the heap to preserve the integrity of the heap.

There are four basic types of memory areas:

- Heap memory is the traditional heap but is managed by the Metronome Garbage Collector.
- Scoped memory that applications must specifically request and can be used only by real-time threads.
- Immortal memory is used by class loading and static initialization even if the application makes no explicit use of it. It represents an area of memory containing objects that can be referenced without exception or garbage collection delay by any schedulable object, specifically including no-heap real-time threads and no-heap asynchronous event handlers.
- Physical memory allows objects to be created within specific physical memory regions that have particular important characteristics, such as memory that has substantially faster access. In general, they are little used and should not affect the standard JVM user at all.

Heap memory

The maximum size is controlled by mx but you should remember NOT to set the initial heap size (**-Xms**) or set it equal to maximum heap size **-Xmx**, because, in real time, the heap never expands from the initial heap size to the maximum heap size; when you have reached maximum heap size with no free space, OutOf MemoryError results. In general, the realtime JVM consumes more heap memory than the traditional JVM because object headers are larger. In addition arrays are broken into fragments each of which has a header. It depends on the ratio of large to small objects and the amount of array usage, but it would not be unexpected to find an application needing 20% more heap space.

The Metronome Garbage Collector is similar to the "mostly concurrent" collector that exists in the mainstream JVM in that it collects garbage while the application is running. In a perfect world, the collection cycle completes before the application runs out of memory but some applications with very high allocation rates can allocate faster than the Metronome Garbage Collector can collect. Various detailed controls affect the collection rate, but there is one control that forces Metronome to revert to traditional stop-the world GC before finally throwing OutOf MemoryError. The runtime parameter is **-Xgc:synchronousGCOnOOM** and the counterpart **-Xgc:nosynchronousGCOnOOM**. The default is **-Xgc:synchronousGCOnOOM** but it is advisable to add the flag to be sure.

Scoped memory

The RTSJ introduces the concept of scoped memory. It can be used by objects that have a well-defined lifetime. A scope may be entered explicitly, or it can be attached to a schedulable object (a real-time thread or an asynchronous event handler) that effectively enters the scope before it executes the object's run() method. Each scope has a reference count and when this reaches zero the objects that are resident in that scope can be closed (finalized) and the memory associated with that scope is released. Reuse of the scope is blocked until finalization is complete.

Scoped memory can be divided into two types: VTMemory and LTMemory. These types of scoped memory vary by the time required to allocate objects from the area. LTMemory guarantees linear time allocation when memory consumption from the memory area is less then the memory area's initial size. VTMemory offers no such guarantee.

Scopes can be nested. When a nested scope is entered, all subsequent allocations are taken from the memory associated with the new scope. When the nested scope is exited, the previous scope is restored and subsequent allocations are again taken from that scope.

Because of the lifetimes of scoped objects, it is necessary to limit the references to scoped objects, by means of a restricted set of assignment rules. A reference to a scoped object cannot be assigned to a variable from an outer scope, or to a field of an object in either the heap or the immortal area. A reference to a scoped object can be assigned only into the same scope or into an inner scope. The virtual machine detects illegal assignment attempts and throws an appropriate exception when they occur. The flexibility provided in choice of scoped memory types allows the application to use a memory area that has characteristics that are appropriate to a particular syntactically defined region of the code.

The size of the area must be specified during construction of the area and a command line parameter, **-Xgc:scopedMemoryMaximumSize**, controls the maximum value. The default is 8 MB and is adequate for most purposes.

Immortal memory

ImmortalMemory is a memory resource shared among all schedulable objects and threads in an application. Objects allocated in Immortal memory are always available to non-heap threads and asynchronous event handlers and are not subject to delays caused by garbage collection. Objects are freed by the system when the program terminates.

The size is controlled by **-Xgc:immortalMemorySize**; for example, **-Xgc:immortalMemorySize=20m** sets 20 MB, The default is 16 MB, which should be adequate unless you are doing a lot of class loading; classloading is probably the cause of most OutOf MemoryError exceptions.

Using memory

A comparison of Java threads, real-time threads, and no-heap real-time threads.

The Real-Time Specification for Java (RTSJ) adds two classes to support real-time threads: RealtimeThread class and NoHeapRealtimeThread class.

- Both real-time threads and no-heap real-time threads are schedulable objects. As schedulable objects, they have the following parameters: release, scheduling, memory, processing group, memory parameters, and logic.
- Real-time threads can access objects in heap memory as well as in scoped and immortal memory.
- No-heap real-time threads access only scoped and immortal memory areas.
- No-heap real-time threads should have a higher priority than other real-time threads. If their priority is less than other real-time threads, they lose their advantage of running without interference from the garbage collector.

Note: A no-heap real-time thread preempts garbage collection when its priority is set higher than a real-time thread that uses the heap for storage.

Table 8. Memory access by real-time and no-heap real-time threads

Threads	Immortal memory	Scoped memory	Heap memory
Normal threads	✓	✗	✓
Real-time threads	✓	✓	✓
no-heap real-time threads	✓	✓	✗

MemoryArea

Immortal memory

Immortal memory is not subject to garbage collection. After the space has been allocated for immortal memory, you cannot change the allocation until the application is closed.

- Because of the nature of immortal memory might wish to find ways to reuse the memory. One possibility is to create a pool of reusable objects. Using scoped memory is an alternative.
- Objects in immortal memory cannot reference anything in scoped memory. If this happens it leads to a IllegalAssignmentError exception.

Scoped memory

Scoped memory can be used as the initial memory area of a schedulable object or may be entered by one. When no longer referenced the area is cleared of all objects. Schedulable objects allocated to scoped memory perform all their functions within that area. Once complete the memory area is closed and the space reclaimed.

Physical memory
Use physical memory when the characteristics of the memory itself are important; for example, non-pageable or non-volatile.

Linear time allocation scheme (LTMemory)
LTMemory represents a memory area guaranteed by the system to have linear time allocation when memory consumption from the memory area is less than the memory area's initial size. Execution time for allocation is allowed to vary when memory consumption is between the initial size and the maximum size for the area. Furthermore, the underlying system is not required to guarantee that memory between initial and maximum will always be available.

The memory area described by a LTMemory instance does not exist in the Java heap and is not subject to garbage collection. Thus, it is safe to use a LTMemory object as the initial memory area associated with a NoHeapRealtimeThread or to enter the memory area using the ScopedMemory.enter() method within a NoHeapRealtimeThread.

Variable time allocation scheme (VTMemory)
VTMemory is similar to LTMemory except that the execution time of an allocation from a **VTMemory** area need not complete in linear time.

HeapMemory
Objects in heap memory cannot reference anything in scoped memory. If they do, an IllegalAssignmentError exception occurs.

Synchronization and resource sharing

In a real-time system, when three or more threads that are running at different priorities and are synchronizing with each other, this can occasionally result in a condition called priority inversion, in which a higher priority thread is blocked from running by a lower priority thread for an extended period of time. WebSphere Real Time avoids this condition using a scheme called priority inheritance.

When a higher priority task is blocked from running by a lower priority task, the lower priority of the lower priority task is temporarily boosted to match the higher priority until the higher priority task is no longer blocked.

Periodic and aperiodic parameters

Real-time threads have a number of release parameters, determining how often a schedulable object is released. Periodic and aperiodic parameters are examples of release parameters.

Periodic parameters

This class is for those schedulable objects that are released at regular intervals.

AbsoluteTime
Is expressed in milliseconds and nanoseconds.

RelativeTime
Is the length of time of a given event expressed in milliseconds and

nanoseconds. For example, you could measure the absolute time when an event starts and finishes. Then calculate the relative time as the difference between the two measurements.

Aperiodic parameters

This class is used by those schedulable objects that are released at irregular intervals. Because a second aperiodic event might occur before the first one has completed, you can define the length of the queue of outstanding requests.

Asynchronous event handling

Asynchronous event handlers react to events that occur outside a thread; for example: input from an interface of an application. In real-time systems, these events must respond within the deadlines that you set for your application.

Asynchronous events can be associated with system interrupts, POSIX signals, and they can be linked to a timer.

Like real-time threads, asynchronous event handlers have a number of parameters associated with them. For a list of these parameters see "Relationships of the real-time threads" on page 30.

Signal Handlers

The POSIXSignalHandler supports the signals SIGQUIT, SIGTERM, and SIGABRT. The default behavior for SIGQUIT causes a javacore to be generated. The generation of the javacore does not interfere with the operation of a running program, apart from execution time and file-IO.

To suppress all core and javacore generation on a failure, use **-Xdump:none**.

To suppress only core and javacore generation on a SIGQUIT, specify **-Xdump:java:none -Xdump:java:events=gpf+abort**.

The following signals can be attached to asynchronous event handlers (AEHs) by the POSIXSignalHandler mechanism (signal descriptions as defined in /usr/include/bits/signum.h):

```
#define SIGQUIT         3       /* Quit (POSIX).  */
#define SIGABRT         6       /* Abort (ANSI).  */
#define SIGKILL         9       /* Kill, unblockable (POSIX).  */
#define SIGUSR1         10      /* User-defined signal 1 (POSIX).  */
#define SIGUSR2         12      /* User-defined signal 2 (POSIX).  */
```

No other signals are currently supported. All of the above are asynchronous signals, and it is impossible to support attaching to synchronous signals (such as SIGILL and SIGSEGV) because they indicate a crash of your application's or the JVM's code, not an externally generated event.

Note: SIGQUIT by default causes the Java application to generate dumps, for example: javacore, when received by the JVM. Although additionally it is delivered to any attached AEH. This might cause confusing or undesirable behavior and can be disabled by using the **-Xdump:none:events=user** option on the Java command line.

Required documentation

WebSphere Real Time implements the Real-Time Specification for Java (RTSJ).

WebSphere Real Time version 1.0 has been certified as RTSJ 1.0.1b compliant against the RTSJ Technology Compatibility Kit 1.0.1 and 1.0.2 version J9 3.0.10 FCS2 and is compliant with the Java Compatibility Kit (JCK) at version 5.0.

Supported facilities

The following facilities are supported:

- Allocation-rate enforcement on heap allocation to limit the rate at which a schedulable object creates objects in the heap.

Unsupported facilities

The following facilities are not supported:

- Priority Ceiling Emulation Protocol. For example, it does not permit PriorityCeilingEmulation to be used as a monitor control policy.
- Atomic access support, except where required for conformance to the specification.
- No schedulers other than the base priority scheduler are available to applications.
- Cost enforcement.

Required documentation for Real-Time Specification for Java

The *Required Documentation* section of the Real-Time Specification for Java (RTSJ) is quoted below. Any deviations from the standard implementation of RTSJ are noted.

1. **The feasibility testing algorithm is the default.**

   "If the feasibility testing algorithm is not the default, document the feasibility testing algorithm."
2. **Only the base priority scheduler is available to applications.**

   "If schedulers other than the base priority scheduler are available to applications, document the behavior of the scheduler and its interaction with each other scheduler as detailed in the Scheduling chapter. Also document the list of classes that constitute schedulable objects for the scheduler unless that list is the same as the list of schedulable objects for the base scheduler."
3. **A schedulable object that is preempted by a higher priority schedulable object will be placed at the front of the queue for its priority.**

   "A schedulable object that is preempted by a higher-priority schedulable object is placed in the queue for its active priority, at a position determined by the implementation. If the preempted schedulable object is not placed at the front of the appropriate queue the implementation must document the algorithm used for such placement. Placement at the front of the queue may be required in a future version of this specification."
4. **Cost enforcement is not supported.**

   "If the implementation supports cost enforcement, then the implementation is required to document the granularity at which the current CPU consumption is updated."
5. **Simple sequential mapping is supported.**

"The memory mapping implemented by any physical memory type filter must be documented unless it is a simple sequential mapping of contiguous bytes."

6. **There are no subclasses for the Metronome Garbage Collector supplied with WebSphere Real Time.**

   "The implementation must fully document the behavior of any subclasses of GarbageCollector."

7. **There are no MonitorControl subclasses supplied with WebSphere Real Time.**

   "An implementation that provides any MonitorControl subclasses not detailed in this specification must document their effects, particularly with respect to priority inversion control and which (if any) schedulers fail to support the new policy."

8. **A schedulable object holding a monitor required by a higher priority schedulable object has its priority boosted to the higher priority until such time as it releases the monitor. If at that point the schedulable object is to be made no longer runnable (that is, there is higher priority work to be done) it will be placed at the back of the queue for its original (unboosted) priority.**

   "If on losing ″boosted″ priority due to a priority inversion avoidance algorithm, the schedulable object is not placed at the front of its new queue, the implementation must document the queuing behavior."

9. **The base scheduler is the only scheduler provided with WebSphere Real Time.**

   "For any available scheduler other than the base scheduler an implementation must document how, if at all, the semantics of synchronization differ from the rules defined for the default PriorityInheritance instance. It must supply documentation for the behavior of the new scheduler with priority inheritance (and, if it is supported, priority ceiling emulation protocol) equivalent to the semantics for the base priority scheduler found in the Synchronization chapter."

10. **The worst case time from the firing of an event to the scheduling of an associated bound event handler will average 40µs and not exceed 100µs providing that there are no competing schedulable objects or system activity of equal or higher priority and providing that garbage collection does not interfere. If the schedulable object driving the fire method, the AsyncEvent object or the handler references the heap then the potential influence of garbage collection is as documented in (B). This assumes that the code is being interpreted and that a single handler (which is bound) is configured on the event.**

    "The worst-case response interval between firing an AsyncEvent because of a bound happening to releasing an associated AsyncEventHandler (assuming no higher-priority schedulable objects are runnable) must be documented for some reference architecture."

11. **The worst case interval between firing an AsynchronouslyInterruptedException at an ATC-enabled thread and the first delivery of that exception will average 35µs and not exceed 160µs providing that there are no competing schedulable objects or system activity of equal or higher priority and providing that garbage collection does not interfere. ATC-enabled in this case means that the thread is executing in an AI enabled method in a region that is not ATC-deferred and those conditions remain true until delivery of the exception. The potential influence of**

garbage collection is as documented in (B). If the target thread is in native code then the delay is potentially unbounded. This assumes that the code is being interpreted.

"The interval between firing an AsynchronouslyInterruptedException at an ATC-enabled thread and first delivery of that exception (assuming no higher-priority schedulable objects are runnable) must be documented for some reference architecture."

12. **Not applicable see response 4.**

    "If cost enforcement is supported, and the implementation assigns the cost of running finalizers for objects in scoped memory to any schedulable object other than the one that caused the scope's reference count to drop to zero by leaving the scope, the rules for assigning the cost shall be documented."

13. **There are no changes to the standard implementation of RealtimeSecurity.**

    "If the implementation of RealtimeSecurity is more restrictive than the required implementation, or has run-time configuration options, those features shall be documented."

14. **The finalizers for objects in a scoped memory area will be run by the last thread to reference that area, that is, they will be run when the thread decrements the reference count from one to zero. Any cost associated with running the finalizers will be assigned to that thread.**

    "An implementation may run finalizers for objects in scoped memory before the scope is reentered and before it returns from any call to getReferenceCount() for that scope. It must, however, document when it runs those finalizers."

15. **The resolution is not settable.**

    "For each supported clock, the documentation must specify whether the resolution is settable, and if it is settable the documentation must indicate the supported values."

16. **There are no other clocks other than the real-time clock provided with WebSphere Real Time.**

    "If an implementation includes any clocks other than the required real-time clock, their documentation must indicate in what contexts those clocks can be used."

Note: **A** The reference architecture for the tests will be an LS20, 4 way, 2GHz with 1Mb cache and 4Gb of memory.

B Garbage collection can cause a delay at any point in a thread which is associated with the heap. The collector can operate in one of two basic modes governing the behavior when heap memory is exhausted. If the collector is set to immediately throw OutOfMemoryError in these circumstances then the worst case garbage collection delay will normally be below 1ms. There are currently some circumstances in which the delay can be higher; for example if there are many threads with deeply nested stacks or large numbers of large sized scopes. If the collector is set to perform a synchronous GC before throwing an OutOfMemoryError then the potential collection delay is related to the number of live objects in the heap and the numbers of objects in other memory areas. In these circumstances the delay should be considered unbounded since it can be many seconds for typical heap sizes.

Chapter 6. The sample application

The sample application uses a series of examples to demonstrate the features of WebSphere Real Time that can be used to improve the real-time characteristics of Java programs.

The source files for the sample application are in the demo/realtime/sample_application.zip file.

The sample consists of two main components:

- **A simulation.** This is a simple example of a lunar lander. Its position is defined by its height above the ground and the distance from the landing area. See Figure 3 on page 42.

  The simulation class is written using no-heap real-time threads (NHRTs) and is not modified any further in this documentation.
- **A controller** that sends commands to the simulation. It sends radar pings to judge the landers height and control the rate of descent of the lander based on this information. The controller also receives a stream of information from the lander, for example, the lander's distance from the landing area.

  The controller is written initially in standard Java. In "Modifying Java applications" on page 51, it is developed into a real-time Java program

Depending on the outcome of the landing, the controller is sent one of two messages; either crash or land.

Using the sample application you can:

- Run both the simulation and the controller together, this demonstrates a combination of real time and standard Java classes running together. For information how to do this, see "Running the sample application" on page 43. This also shows the output that you can expect from the sample application.

  Note: You can start them both at the same time using the LaunchBoth class.
- Compare the difference when using the Metronome Garbage Collector and the standard Garbage Collector. For information how to do this, see "Running the sample application without Real Time" on page 43 and "Running the sample application with Metronome" on page 44.
- Run the application using the ahead-of-time (AOT) complier. For information how to do this, see "Running the sample application using AOT" on page 45.

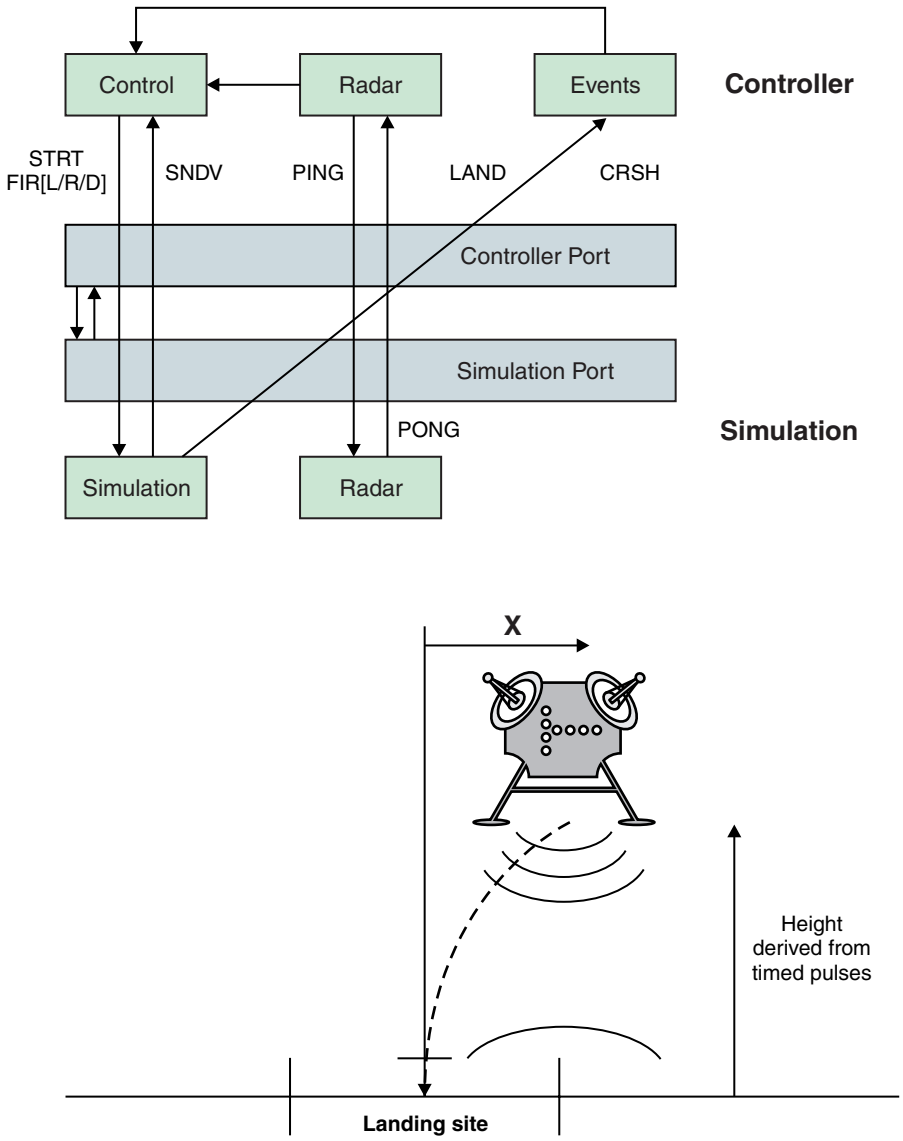

Figure 3. Diagram of the lunar lander

Building the sample application

The sample application source code is provided for guidance. Preparation requires unpacking and compilation of the Java source code before it can be run.

1. Create a working directory.
2. Extract the sample application into your working directory.

   ```
   unzip sample_application.zip
   ```

3. Create a new directory for your output.

   ```
   mkdir classes
   ```

4. Compile the source.
   a. Generate a list of the files.

      ```
      find -name "*.java" > source
      ```

b. Compile the source.

```
javac -Xrealtime -g -d classes @source
```

c. Create a jar file of the class files.

```
jar cf demo.jar -C classes/
```

You should now be able to run the sample application.

Running the sample application

WebSphere Real Time provides a standard JVM as well as a Real Time JVM, invoked with the **-Xrealtime** command-line argument.

The sample application has two components, designed to be run in separate JVMs:

- The Simulation, which only runs in Real Time Java.
- The Controller, which can run either non-Real Time or Real Time Java.

Running the Controller code in a variety of modes demonstrates the benefits of the IBM Real Time Java technology.

Running the sample application without Real Time

In this procedure you run the sample application without taking advantage of IBM WebSphere Real Time.

To run the sample application, you must first build the sample source code. See "Building the sample application" on page 42 for more information.

1. Start the simulation.

   ```
   java -Xrealtime -cp ./demo.jar -Xgc:scopedMemoryMaximumSize=11m demo.sim.SimLauncher <port>
   ```

   where <*port*> is an unallocated port for this machine.

2. Start the controller.

   ```
   java -cp ./demo.jar -verbose:gc -mx300m demo.controller.JavaControlLauncher <host> <port>
   ```

   where <*host*> is the hostname of the machine running the simulation and <*port*> is the port specified in the previous step.

The output of this application produces the following message showing that the simulation and the controller have started.

```
SimLauncher: Waiting for connections...
Starting control thread...
```

Some point samples of the values in the controller are printed out to the console:

```
x=99.50, radar=199.11, y=198.34, vx=-0.71, vy=-0.43, timeSinceLast=0.19, targetVx=-6.01, targetVy=-9.00
x=95.50, radar=194.59, y=192.70, vx=-2.70, vy=-2.43, timeSinceLast=0.20, targetVx=-5.94, targetVy=-9.00
x=87.50, radar=186.57, y=183.06, vx=-4.70, vy=-4.40, timeSinceLast=0.20, targetVx=-5.77, targetVy=-9.00
x=76.46, radar=172.84, y=169.42, vx=-5.42, vy=-6.75, timeSinceLast=0.20, targetVx=-5.60, targetVy=-9.00
x=65.36, radar=155.58, y=151.84, vx=-5.50, vy=-9.19, timeSinceLast=0.20, targetVx=-5.57, targetVy=-9.00
x=54.36, radar=138.06, y=135.24, vx=-5.44, vy=-7.63, timeSinceLast=0.20, targetVx=-5.56, targetVy=-9.00
x=43.26, radar=120.57, y=117.22, vx=-5.67, vy=-9.62, timeSinceLast=0.20, targetVx=-5.52, targetVy=-9.00
x=32.36, radar=103.60, y=100.72, vx=-5.47, vy=-9.06, timeSinceLast=0.20, targetVx=-5.43, targetVy=-9.00
x=21.52, radar=84.60, y=82.86, vx=-5.32, vy=-9.09, timeSinceLast=0.20, targetVx=-5.60, targetVy=-9.00
x=10.72, radar=67.07, y=65.56, vx=-5.30, vy=-10.54, timeSinceLast=0.20, targetVx=-5.65, targetVy=-9.00
x=0.76, radar=51.08, y=49.78, vx=-4.30, vy=-7.52, timeSinceLast=0.20, targetVx=-0.50, targetVy=-9.00
x=-5.24, radar=37.07, y=35.94, vx=-2.30, vy=-8.26, timeSinceLast=0.20, targetVx=0.50, targetVy=-9.00
x=-7.24, radar=20.05, y=19.90, vx=-0.30, vy=-6.15, timeSinceLast=0.20, targetVx=0.50, targetVy=-9.00
x=-6.36, radar=2.68, y=2.80, vx=0.27, vy=-10.08, timeSinceLast=0.20, targetVx=0.50, targetVy=-9.00
```

Just before the simulation stops, the following message is issued:

```
Fire down transitions 141, fire horizontally transitions 141
LAND!
```

In addition to this, the controller produces a graph called graph.svg in the same directory. Figure 4 shows the effect of garbage collection pauses on the JavaRadar thread with a standard non-Real Time JVM. The spikes in the blue Radar Height line show how standard garbage collection pauses affect the Controller application. On some runs, the garbage collection pauses are long enough to can cause failures, leading to the message:

```
CRASH!
```

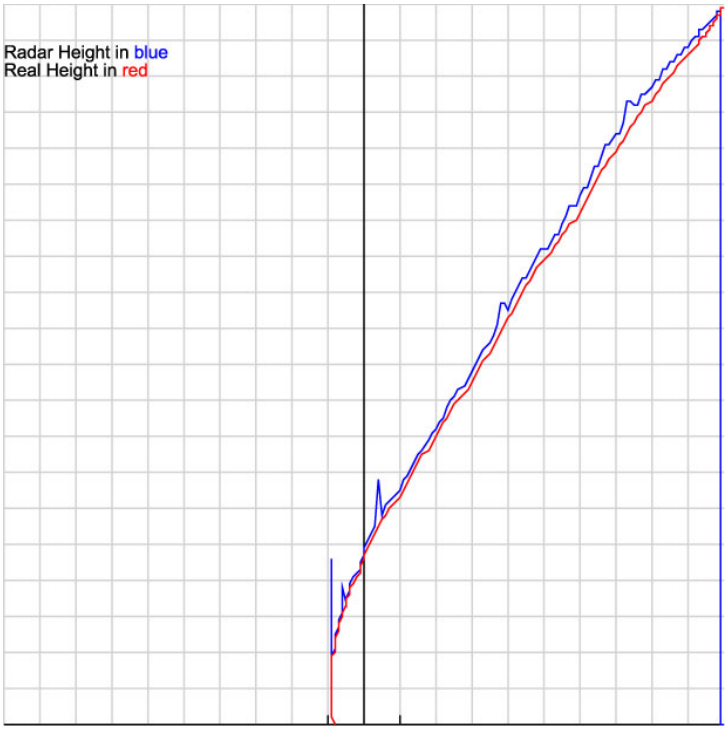

Figure 4. Graph generated by the sample application without Real Time

Running the sample application with Metronome

You can run a standard Java application in a real-time environment without any need to rewrite the code, by adding the **-Xrealtime** option.

This enables both Real Time Java language features and the Metronome Garbage Collector.

To run the sample application, you must first build the sample source code. See "Building the sample application" on page 42 for more information.

1. Start the simulation.

   ```
   java -Xrealtime -cp ./demo.jar -Xgc:scopedMemoryMaximumSize=11m demo.sim.SimLauncher <port>
   ```

   where <*port*> is an unallocated port for this machine.
2. Start the controller.

   ```
   java -Xrealtime -cp ./demo.jar -verbose:gc -mx300m demo.controller.JavaControlLauncher <host> <po
   ```

where <*host*> is the hostname of the machine running the simulation and <*port*> is the port specified in the previous step. Running both JVMs on the same machine can lead to less deterministic behavior. See "Considerations" on page 4 for more information.

As with the non-Real Time example, the application will now run and generate data points and a graph.

In this run with Metronome, the graph shows no spikes in the blue Radar Height, and very accurate tracking of the red Real Height line, because the Controller code is now running with only very short garbage collection pauses.

The difference between the short, multiple Metronome Garbage Collection pauses (typically less than 1 millisecond) and the longer, fewer pauses of non-Real Time garbage collection (typically 10s or 100s of milliseconds) can be seen by adding the **-verbose:gc** option to the Controller run command.

See "Using the -verbose:gc option" on page 23 for more information on the verbose garbage collection output.

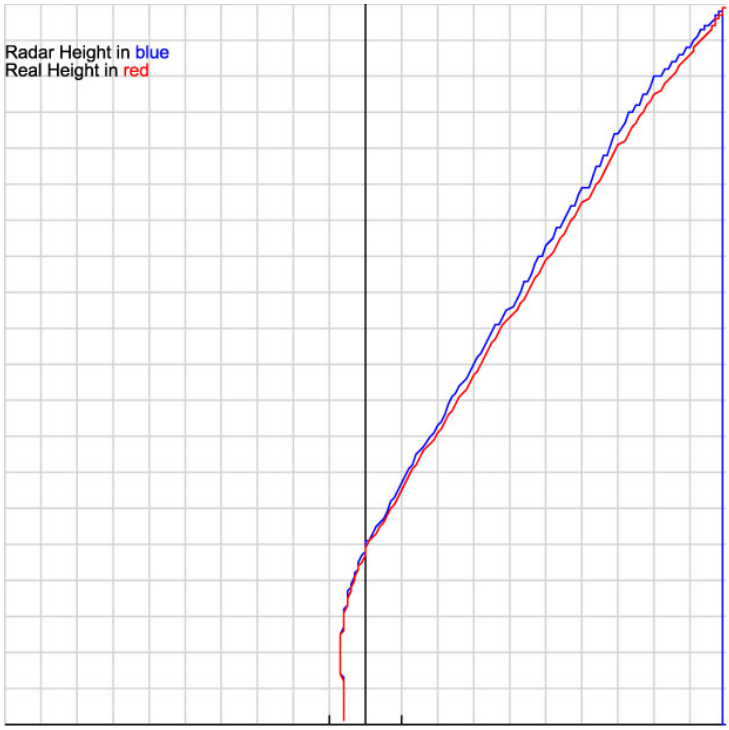

Figure 5. Graph generated by the sample application with Real Time

Running the sample application using AOT

This procedure runs a standard Java application in a real-time environment using the ahead-of-time compiler, without the need to rewrite code. Use this to compare running the same application using the JIT compiler.

See Chapter 3, "Using the ahead-of-time compiler and WebSphere Real Time," on page 15 for a more detail on ahead-of-time compilation.

To run the sample application, you must first build the sample source code. See "Building the sample application" on page 42 for more information.

The ahead-of-time (AOT) compiler allows you to compile your Java application to native code before running it. This enables you to predict how the application runs more precisely, because there will be no interruptions caused by just-in-time (JIT) compilation.

1. Convert the application bytecodes into native code.
   a. Identify which bytecodes to pre-compile by running the sample with the normal JIT compiler.

      ```
      java -Xrealtime -Xjit:verbose={precompile},vlog=./sim.aotOpts \
           -cp ./demo.jar -Xgc:scopedMemoryMaximumSize=11m demo.sim.SimLauncher <port>
      ```

      And in a separate window:

      ```
      java -Xrealtime -Xjit:verbose={precompile},vlog=./control.aotOpts \
           -cp ./demo.jar -Xmx300m demo.controller.JavaControlLauncher localhost <port>
      ```

      where *<port>* is an unallocated port for this machine. You will see messages similar to the following:

      ```
      Fire down transitions 141, fire horizontally transitions 141
      ```

      and,

      ```
      Land!
      ```

      Then exit the SimLauncher process.
   b. Combine the AOT options files.

      ```
      cat sim.aotOpts control.aotOpts > sample.aotOpts
      ```
   c. Create an output directory for the AOT demo.jar file.

      ```
      mkdir aot
      ```
   d. Compile the files in the sample.aotOpts file in realtime.jar and vm.jar.

      ```
      jxeinajar -Xrealtime -searchPath $JAVA_HOME/jre/bin/realtime/jclSC150/ \
                -outPath aot -optFile sample.aotOpts
      ```
   e. Compile the files in the sample.aotOpts file in core.jar.

      ```
      jxeinajar -Xrealtime $JAVA_HOME/jre/lib/core.jar \
                -outPath aot -optFile sample.aotOpts
      ```
   f. Compile the sample application classes.

      ```
      jxeinajar -Xrealtime -outPath ./aot demo.jar
      ```

      You will see the following output:

      ```
      J9 Java(TM) jxeinajar 2.0
      Licensed Materials - Property of IBM

      (c) Copyright IBM Corp. 1991, 2006  All Rights Reserved
      IBM is a registered trademark of IBM Corp.
      Java and all Java-based marks and logos are trademarks or registered
      trademarks of Sun Microsystems, Inc.

      Found /$APPDIR/rtdemos/demo.jar
      Converting files
      Converting /$APPDIR/rtdemos/demo.jar into /$APPDIR/rtdemos/aot//demo.jar
      JVMJ2JX002I Precompiled 160 of 162 method(s) for target ia32-linux.
      Succeeded to JXE jar file demo.jar

      Processing complete

      Return code of 0 from jxeinajar
      ```

      The demo application is now ready to be run using ahead-of-time compilation.

2. Start the simulation.

```
java -Xrealtime -Xnojit -Xmx300m \
     -Xbootclasspath/p:aot/core.jar:aot/vm.jar:aot/realtime.jar \
     -cp $APPDIR/aot/demo.jar -Xgc:scopedMemoryMaximumSize=11m \
     demo.sim.SimLauncher <port>
```

   where <*port*> is an unallocated port for this machine.

3. Start the controller.

```
java -Xrealtime -Xnojit -Xmx300m \
     -Xbootclasspath/p:aot/core.jar:aot/vm.jar:aot/realtime.jar \
     -cp $APPDIR/aot/demo.jar \
     demo.controller.JavaControlLauncher <host> <port>
```

   where <*host*> is the hostname of the machine running the simulation and <*port*> is the port specified in the previous step. Running both JVMs on the same machine can lead to less deterministic behavior. See “Considerations” on page 4 for more information.

As with the non-Real Time example, the application will now run and generate data points and a graph.

In this run with ahead-of-time compilation, the graph shows no spikes in the blue Radar Height, and very accurate tracking of the red Real Height line, because the Controller code is now running with only very short garbage collection pauses and no just-in-time compilation interruptions.

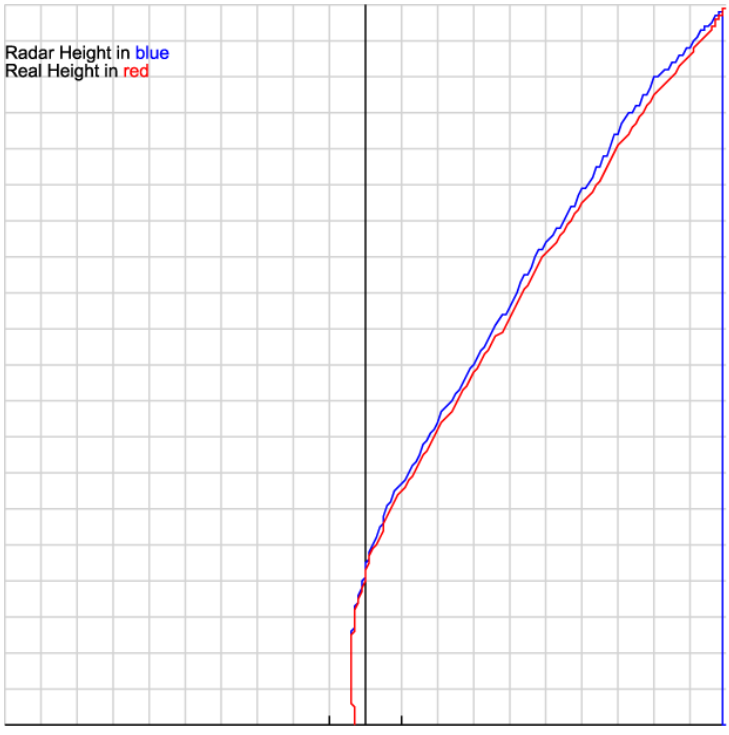

Figure 6. Graph generated by the sample application with ahead-of-time compilation

Chapter 7. Writing Java applications to exploit real time

These examples describe how to exploit the real-time environment. They range from the simplest example, running a Java application in real time without any modifications to the code, through to a more complex process of planning and writing no-heap real-time threads. Reasons are provided to help you decide which approach might be most suitable for your applications.

Introduction to writing real-time applications

You do not have to write elaborate no-heap real-time applications to exploit the features of real-time technology. Some of the benefits can be used with very little change to your existing code.

For application programers, here are the steps that you can take to exploit WebSphere Real Time.

1. Initially, you can run a standard Java application in a real-time JVM to give you the benefit of Metronome garbage collection and achieve significant improvement in the predictability of the run time of your application.
2. Add the **-Xnojit** option after you have precompiled your code to use the ahead-of-time (AOT) compiler. See "Building precompiled files" on page 17.
3. Replace java.lang.Thread with javax.realtime.RealtimeThread in your application. There might be slight improvement when compared with the AOT option.

   The main advantage of using real-time threads is the ability to control the priority that you give to each of the threads. Real-time threads can also be made periodic. To exploit these advantages, you must in general be prepared to make changes to the application itself.
4. Plan and write a specific application to use real-time threads and asynchronous event handlers to deal with timers or external events. There are three factors that you need to consider:
   - Planning the priority that you should assign to your real-time threads and
   - Deciding which memory areas you will use to hold objects
   - Communicating with the event handlers.
5. Plan and write a specific application to use no-heap real-time threads. No-heap real-time threads are extensions of real-time threads and you have to consider the priority you assign as well and the memory area. In general, this step should only be taken if the application must handle events in times comparable to the GC pause time (sub millisecond). The complexity of developing with no-heap real-time threads should not be underestimated.

Figure 7 on page 50 shows the steps described above.

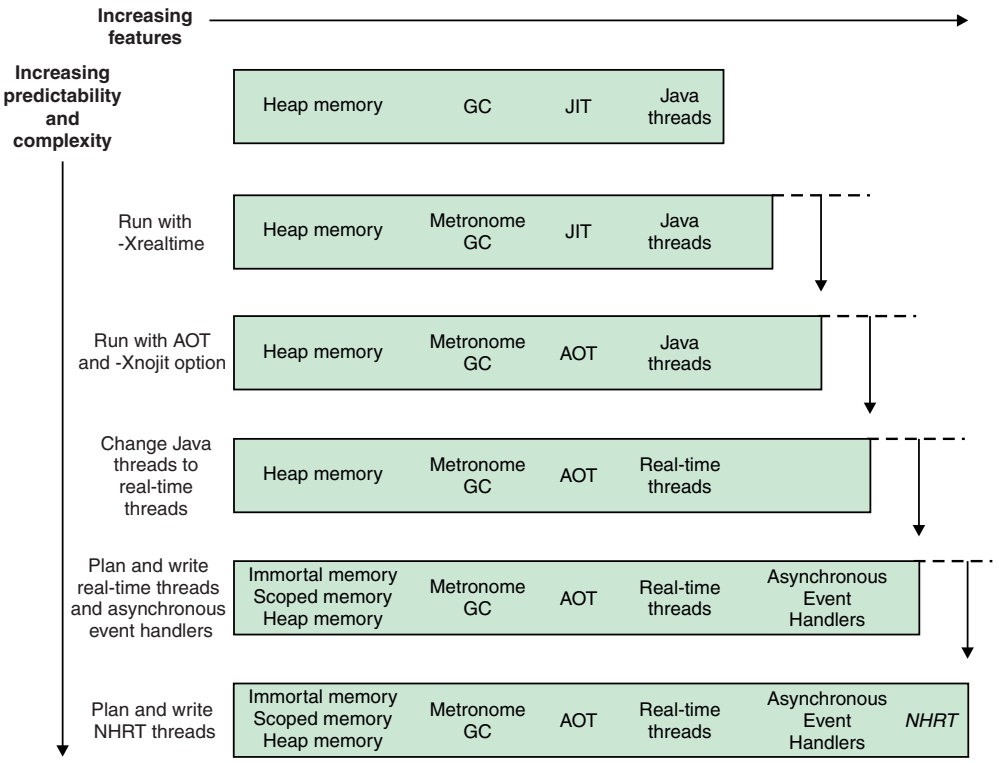

Figure 7. A comparison of the features of RTSJ with the increased predictability.

Planning your WebSphere Real Time application

When you are preparing to write real-time Java applications, you must consider whether to use Java threads, real-time threads, or no-heap real-time threads. In addition, you can decide which memory area that your threads will use.

When planning your application, these steps describe the decisions you need to make:

1. Identify your tasks.
2. Decide the timing periods:
   - Responses greater than 10ms, choose Java threads, just exploiting the Metronome Garbage Collector.

     These threads use only the heap memory for storage. Their disadvantage is that garbage collection interrupts your application but, because it is controlled by the Metronome Garbage Collector, the length and timing of the interruptions are predictable.
   - Responses less than 10ms choose real-time threads.

     Real-time threads can be placed in heap, scoped, or immortal memory. The benefits of using real-time threads are:
     – They can run at a higher priority than standard Java threads.
     – Garbage collection is under the control of the Metronome. However, the garbage collector runs at a higher priority than the highest priority of a real-time thread and interrupts the execution of your program.
   - Responses less than a millisecond, choose no-heap real-time threads.

The priority of no-heap real-time threads can be set higher than garbage collection and therefore is not interrupted significantly by the Metronome. Only the Metronome alarm thread runs at the top level of priority and that uses very small amounts of CPU.

3. Your application might require asynchronous event handlers. This requirement depends on the structure of your program.
   - A time response less than 10ms, choose real-time threads.
   - A time response less than a millisecond, choose no-heap real-time threads
4. Determine thread priorities. In general, the shorter the time period; the higher the priority.
5. Decide memory characteristics.
   - If a task has a variable or high allocation rate which might overwhelm the GC, consider imposing a rate limit (using MemoryParameters) or consider allocating into a scoped memory area.
   - If a task generates a large amount of temporary data during a calculation, consider using a scoped memory area.
   - If a task generates some data during startup that is required for the lifetime of the JVM, consider using immortal memory. Try to avoid using immortal memory in cases where object will continue to be created over the life of the JVM.
   - If tasks need to communicate, particularly if one is running under a no-heap real-time thread, consider using a scoped memory area for the communication.
   - If a task is running under a no-heap real-time thread, consider building a scoped memory area, for example LTMemory, to contain the no-heap thread, the runtime parameters, and possibly the wait-free queues which are used to communicate with the task. The LTMemory object should be built either in immortal or another scope to avoid errors when the no-heap thread attempts to reference it.
6. When you have decided the structure and content of your application, the next steps describe how to modify the runtime options to improve the performance of your application.
   a. During initial testing of your application, set generous amounts of space in heap, scoped, and immortal memory using the **-Xmx**, **-Xgc:immortalMemorySize=size** and **-Xgc:scopedMemoryMaximumSize=size** options.

      Note: With the Metronome GC the initial and maximum heap sizes must be the same because Metronome GC does not increase the size of the heap. Growing the heap is a non-deterministic operation.
   b. Use the **-verbose:gc** option to determine the amount of memory used.
   c. Modify the **-Xgc:targetUtilization** option to allow sufficient time for garbage collection to occur. The default is 70% and this percentage should be adequate for most applications. Ensure that the garbage collection rate is slightly higher than the allocation rate.
   d. Set a realistic size for heap memory using the **-Xmx** option.

Modifying Java applications

To write code that makes use of the real-time Java features, use javax.realtime.RealtimeThread to replace java.lang.Thread for threads.

This example is based on JavaRadar.java class found in the demo/realtime/sample_application.zip file.

The programming model for real-time threads is similar to that for standard Java applications. However, this rather crude way of adding real-time threads to your programs does not take full advantage of the features of WebSphere Real Time. To do so you would need to modify the threads so that they had a priority associated with them and also consider what memory areas they should be using.

Just by changing the classes of your threads, you gain only a slight benefit for your application because the default priority of a real-time threads is greater than that of standard Java threads.

To change JavaRadar to a RealtimeThread, you change the class it extends from Thread to RealtimeThread.

Replacement of java.lang.Thread by javax.realtime.RealtimeThread

The class in the sample application called JavaRadar extends java.lang.Thread. For example:

```
public class JavaRadar extends Thread implements Radar
```

To make this Java thread a real-time thread, you redefine this class definition as follows:

```
public class RTJavaRadar extends RealtimeThread implements Radar
```

Writing real-time threads

So far you have just modified an application; now it is time to write some code. You can write applications that utilize real-time threads to take advantage of the real-time priority levels and memory areas.

This example is based on JavaRadar.java, RTJavaRadar.java, and RTJavaControlLauncher.java classes found in the demo/realtime/sample_application.zip file.

This sample shows you how to use immortal memory with the same sample that is described in “Modifying Java applications” on page 51.

The programming model for real-time threads is similar to that for standard Java applications.

The benefits of using real-time threads are:

- Full support for OS-level thread priorities on real-time threads.
- The use of scoped or immortal memory areas.
  - With scoped memory you can explicitly control the deallocation of memory without impacting Garbage Collection.
  - With no-heap real-time threads, you can use immortal memory to avoid garbage collection pauses.
  - Those real-time threads that reference objects in the heap are subject to garbage collection as are those real-time threads that reside in the heap memory.
  - No-heap real-time threads cannot reference objects in heap memory and, as a consequence, they are not impacted by garbage collection.

In Table 9 the priorities are assigned on the basis that the SimulationThread has the highest priority because it represents external events and should not be allowed to be preempted by anything within the program. The RadarThread needs to respond quickly to the pings from the controller. The quicker the response the more accurate the measurement of the lunar lander's height can be. The ListenThread also has to respond quickly to commands from the controller but takes second place to the RadarThread.

These three threads are in scoped memory because the simulation runs as a server. After it has run a simulation, it can exit the scoped memory area and then reenter it to wait for another run of the simulation. It is using scoped memory so that it can reset itself.

RTJavaRadarthread has the highest priority of the controller threads because it is more sensitive to timing because it is using this time to derive the height. It is immortal because it is running as a NHRT and the controller is run only once and the memory is released when the JVM exits.

For RTJavaControlThread and RTJavaEventThread, the time constraints are not that critical and therefore using heap is acceptable.

Finally, RTLoadThread performs no useful function for the lunar lander. However, it demonstrates that significant memory allocation and deallocation can be performed at a lower priority than other threads and not impact the performance of the higher priority threads of the lunar lander.

Table 9. Relationship of threads to memory areas in the sample application

Memory	Thread	Priority
Scoped	demo.sim.SimulationThread	38
	demo.sim.RadarThread	37
	demo.sim.SimulationThread.ListenThread	36
Immortal	demo.controller.RTJavaRadarThread	15
Heap	demo.controller.RTJavaControlThread	14
	demo.controller.RTJavaEventThread	13
Scoped and Heap	demo.controller.RTLoadThread	12

Examples

This code from demo.sim.SimulationThread shows where the priority of 38 has been set. **1** This line of code retrieves the maximum priority that is available in the JVM.

```
super(null, area);

// Set priority separately, as we are using "this".
// Note that PriorityScheduler.MAX_PRIORITY has been deprecated.
this.setSchedulingParameters(new PriorityParameters(PriorityScheduler
        .getMaxPriority(this)));     1
```

This code from demo.sim.SimLauncher shows where scoped memory has been defined. **2** shows the allocation of LTMemory which is a scoped memory area that allocates memory in linear time.

```
final IndirectRef<MemoryArea> myMemRef = new IndirectRef<MemoryArea>();

/*
```

```
 * The LTMemory object has to be created in a memory area that the
 * NHRTs can access.
 */
ImmortalMemory.instance().enter(new Runnable() {
    public void run() {
        myMemRef.ref = new LTMemory(10000000); 2
    }
});

final MemoryArea simMemArea = myMemRef.ref;
```

The ScopedMemoryArea object referenced by “simMemArea” is being allocated within immortal memory, because the NHRT must be able to reference the object that represents the ScopedMemoryArea. Allocating it on the heap would result in the NHRT constructor throwing an IllegalArgumentException, because its memory area argument was on the heap.

```
simMemArea.enter(new Runnable() {
    public void run() {
        try {
            CommsControl commsControl = new CommsControl();
```

This code from demo.controller.RTJavaControlLauncher shows where immortal memory has been defined and used by RTJavaRadar. Because RTJavaRadar runs once during the whole lifetime of the controller JVM, it is designed to allocate memory only on startup; it can be safely run in immortal memory. The design of the application benefits from this because the Controller can access the RTJavaRadar methods without having to first enter the scoped memory area. This would be difficult because Controller was written to run in ordinary Java as well as real-time Java.

```
final RadarPort radarPort = commsControl.getRadarPort();
EventPort eventPort = commsControl.getEventPort();

final IndirectRef<RTJavaRadar> radarRef = new IndirectRef<RTJavaRadar>();

// Create RTJavaRadar in Immortal, it is an NHRT.
// If it was in scoped, it's interaction with the other threads would
// be more complex.
ImmortalMemory.instance().enter(new Runnable() {
    public void run() {
        // Realtime version of Radar.
        radarRef.ref = new RTJavaRadar(radarPort, ImmortalMemory
                .instance());
    }
});

RTJavaRadar radarJava = radarRef.ref;
```

Writing asynchronous event handlers

Asynchronous event handlers react to timer events or to events that occur outside a thread; for example, input from an interface of an application. In real-time systems, these events must respond within the deadlines that you set for your application.

This example is based on RTJavaEventThread.java and RTJavaControlLauncher.java classes found in the demo/realtime/sample_application.zip file.

In the sample application, the event thread waits on events from the simulation that signals a crash or a landing. In the real-time version of this thread, the AsyncEvent mechanism is used. These events are used to print out the appropriate status message and to cause the controller to exit.

The RTJava EventThread has two asynchronous events defined. They both have no parameters.

```
public class RTJavaEventThread extends RealtimeThread {
⋮
    private AsyncEvent landEvent = new AsyncEvent(),   Land
            crashEvent = new AsyncEvent();  Crash
```

These events create and register two asynchronous event handlers.

```
    /**
     * Pass a runnable object that will be fired when the land event occurs.
     *
     * @param runnable code to be executed when land event is triggered.
     */
    public void addLandHandler(Runnable runnable) {
        AsyncEventHandler handler = new AsyncEventHandler(runnable);
        this.landEvent.addHandler(handler);
    }

    /**
     * Pass a runnable object that will be run when the crash event occurs.
     *
     * @param runnable code to be executed when crash event is triggered.
     */
    public void addCrashHandler(Runnable runnable) {
        AsyncEventHandler handler = new AsyncEventHandler(runnable);
        this.crashEvent.addHandler(handler);
    }
```

When the crash or land messages are received, their corresponding asynchronous event handler is fired, causing their asynchronous event handler's Runnables to be released.

```
⋮
                tag = this.eventPort.receiveTag();

                switch (tag) {
                case EventPort.E_CRSH:
                    // Crash
                    this.crashEvent.fire();
                    this.running = false;
                    break;
                case EventPort.E_LAND:
                    // Land
                    this.landEvent.fire();
                    this.running = false;
                    break;
                }
```

In RTJavaControlLauncher.java, there are invocations of the addLandHandler and addCrashHandler methods. The Runnables passed cause a message to be printed onto the console and the control thread is stopped when their associated asynchronous event handlers are fired. See RTJavaEventThread.java for the point where they are triggered.

```
            // AEH runnable for land handler.
            javaEventThread.addLandHandler(new Runnable() {
                public void run() {
                    System.out.println("LAND!");
                }
            });

            // AEH runnable for crash handler.
            javaEventThread.addCrashHandler(new Runnable() {
```

```
            public void run() {
                System.out.println("CRASH!");
            }
        });
```

Writing NHRT threads

This tutorial describes how to add no-heap real-time threads (NHRT) to a Java application. Using this tutorial you can begin to develop or modify your own programs.

This example is based on SimulationThread.java and SimLauncher.java classes found in the demo/realtime/sample_application.zip file.

The demo.sim.SimulationThread class is part of the simulation in the demo application. It is intended to act as a substitute for the real world, and therefore, should run without interruption from the rest of the system. To achieve this the thread is created as a NoHeapRealtimeThread with the highest available priority. This ensures that the thread is not interrupted by garbage collection or by other threads on the system.

In SimulationThread the following constructor calls the super constructor "NoHeapRealtimeThread(SchedulingParameters scheduling, MemoryArea area)", before then setting its SchedulingParameters and ReleaseParameters separately.

```
public SimulationThread(MemoryArea area, ControlPort controlPort,
            EventPort eventPort, RadarThread radarThread) {

        super(null, area);

        // Set priority separately, as we are using "this".
        // Note that PriorityScheduler.MAX_PRIORITY has been deprecated.
        this.setSchedulingParameters(new PriorityParameters(PriorityScheduler
                .getMaxPriority(this)));

        ReleaseParameters releaseParms = new PeriodicParameters(null,
                new RelativeTime(period, 0)); // 20ms cycle (50Hz)
        this.setReleaseParameters(releaseParms);

        // It is good practice to identify each of the threads.
        this.setName("SimulationThread");

        this.controlPort = controlPort;
        this.eventPort = eventPort;
        this.radarThread = radarThread;
    }
```

The other active threads in the simulation are also created as no-heap real-time threads (NHRTs), but of slightly lower priority. See "Writing real-time threads" on page 52 for the arrangement of the priorities.

The simulation has the option of running indefinitely, such that once a simulation has completed it will restart. As the simulation is composed of NHRTs there is the choice of using ScopedMemory or ImmortalMemory. The sample application uses ScopedMemory for the Simulation as it is appropriate to exit the ScopeMemoryArea that was allocated when the simulation finished and then reenter it to wait for the next run. In this case, no state is carried over from one run to the next.

Most classes are NHRT safe, however, most classes can be executed in a manner that is not NHRT safe. For example, if the DatagramSockets were kept in

ImmortalMemory, or in an outer scoped memory area, there would be problems as they are not designed to span memory areas. The sample application uses just the one ScopedMemory area to prevent this.

Memory allocation in RTSJ

In RTSJ there are a number of ways of allocating an object in a specific memory area and it is not always obvious which of these should be chosen at any given point.

Each approach has some characteristics which vary between implementations of RTSJ and make a difference to either the performance or the eventual memory footprint. This section outlines the available options and suggests occasions where they might be the most appropriate choice for allocating an object.

Static initializer

The simplest way to allocate an object in the immortal memory area is to allocate it in a static initializer. The advantage is that you do not have to deal with the issues of changing memory context but the circumstances where this pattern is appropriate are quite limited. This approach is efficient in that the amount of immortal memory consumed is limited to that required for the object itself.

MemoryArea.newInstance(Class c)

This is a simple approach if a thread is in some memory context and wants to allocate an object in some other area which must already be in the scope stack of the thread. The advantage is that you only need access to the class to be instantiated, but this means that the newInstance method must build an appropriate constructor. This pattern is most appropriate if objects of a given class must be allocated infrequently but otherwise tends to show high memory overhead.

MemoryArea.newInstance(Constructor c, Object[] args)

Again a simple approach if a thread is in some memory context and wants to allocate an object in some other context which must already be in the scope stack of the thread. In this case you must pass a Constructor and some arguments and thus takes on the responsibility of ensuring that Constructor is valid in the current memory context. Since the newInstance method does not have to build a Constructor; the overhead is lower than newInstance(Class c) and thus this pattern is more appropriate if objects are to be allocated more frequently and you are willing to pay the price of allocating the constructor up front and storing it somewhere such as ImmortalMemory.

MemoryArea.enter(Runnable r) followed by new <class>()

This approach makes the given MemoryArea the new default for allocations and removes the need for reflection and the attendant Constructor objects. Hence it is most appropriate if many objects are to be created since there is no overhead above the object itself. This approach only works if the desired area is not already active; in the scope stack of any thread. The need to build a Runnable makes this approach more complex than using newInstance since you generally need to pass parameters either on the Runnable or through static or instance fields.

MemoryArea.executeInArea(Runnable r) followed by new <class>()

Again this approach makes the given MemoryArea the new default for allocations and removes the need for reflection and the attendant Constructor objects. Hence it is most appropriate if many objects are to be created since there is no overhead above the object itself. This approach can be used if the desired area is already in the scope stack of the current thread and hence is more flexible than MemoryArea.enter. The need to build a Runnable makes this approach more complex than using newInstance since you generally need to pass parameters either on the Runnable or through static or instance fields.

Class.newInstance()

This approach builds the new instance in the current memory area and thus must be used with either MemoryArea.enter or executeInArea. There is no memory overhead above the object itself.

Example

Table 10 shows the memory required to allocate a single object in each of the above ways. The object being allocated is either an object or a subclass that adds no fields.

Note: There is a significant difference between the cases of allocating an instance of a public class, a package private class or a static inner class. This difference is caused by the significant memory allocation that is done during access checking.

Table 10. Memory requirements for a type of object allocated by different methods (in bytes)

	public system class (java.lang.Object)	public class in package	static inner class
Static initializer	16	16	16
newInstance(Class c)	216	216	3628
newInstance(Constructor c, Object[] args	120	140	140
MemoryArea.enter()	16	16	16
MemoryArea.executeInArea()	16	16	16
Class.newInstance	16	16	16

Using the high resolution timer

The real-time clock provides more precision that the clocks associated with the standard JVM.

This example is based on RTJavaRadar.java class found in the demo/realtime/sample_application.zip file.

Ordinary Java has limited ability for dealing with clocks and timers. The Real-Time Specification for Java allows absolute times to be specific with nanosecond precision and sufficient magnitude for wall-clock time. javax.realtime.HighResolutionTime and its subclasses are used to represent time with two components, milliseconds and nanoseconds.

WebSphere Real Time uses the support of the underlying operating system to supply the high resolution time. Current® Linux kernels supply a clock with, at best, 4 millisecond guaranteed precision. The Linux patches supplied with WebSphere Real Time provide a clock with a precision of closer to 1 microsecond.

The RTJavaRadar class demonstrates the use of the high resolution timer.

- 1 gets the real-time clock
- 2 gets the current absolute time
- 3 gets the nanosecond component of time. The accuracy of the real-time clock means that using nanoseconds is reasonable.
- 4 gets the time before and after the ping.
- 5 returns the speed of descent of the lander.
- 6 The thread then waits for 5 milliseconds before performing another iteration.

```
public void run() {
       // The following objects are created in advance and reused each
       // iteration.
       Clock rtClock = Clock.getRealtimeClock();         1
       AbsoluteTime time = rtClock.getTime();            2

       try {
           double height = 0.0, lastheight;
           long millis = time.getMilliseconds(), lastmillis;
           long nanos = time.getNanoseconds(), lastnanos;     3

           while (this.running) {

               lastmillis = millis;
               lastnanos = nanos;
               lastheight = height;

               // Rather than use the time = rtClock.getTime() form, this
               // method
               // replaces the values in a preexisting AbsoluteTime object.
               rtClock.getTime(time);                 4
               millis = time.getMilliseconds();
               nanos = time.getNanoseconds();

               // We time the time it takes to send the ping and receive the
               // pong.
               this.radarPort.ping();

               rtClock.getTime(time);                 4

               height = (time.getMilliseconds() - millis)
                       / demo.sim.RadarThread.timeScale;
               height += ((time.getNanoseconds() - nanos) / 1.0e6)       5
                       / demo.sim.RadarThread.timeScale;

               double difference = ((double) (millis - lastmillis)) / 1.0e3
                       + ((double) (nanos - lastnanos)) / 1.0e9;
               double speed = (height - lastheight) / difference;

               this.myHeight = height;
               this.mySpeed = speed;

               try {
                   sleep(5);                     6
               } catch (InterruptedException e) {
                   // This is not important.
               }
           }
```

The preceding code can be compared with the following the standard JVM code in the JavaRadar class:

```
public void run() {
        try {
            double height = 0.0, lastheight;

            long nanos = System.nanoTime(), lastnanos;
            while (this.running) {
                /* Set the height every x milliseconds */
                Thread.sleep(5);
                lastnanos = nanos;
                lastheight = height;

                nanos = System.nanoTime();

                this.radarPort.ping();

                // Time scale is height units per millisecond
                height = ((System.nanoTime() - nanos) / 1.0e6)
                        / demo.sim.RadarThread.timeScale;

                double speed = (height - lastheight)
                        / (((double) (nanos - lastnanos)) / 1.0e9);

                this.myHeight = height;
                this.mySpeed = speed;
            }
```

Chapter 8. Using no-heap real-time threads

While Metronome garbage collection provides more consistent response times, sometimes it is appropriate to avoid interruptions from garbage collection altogether.

NoHeapRealtimeThreads (NHRTs) are an extension to **RealtimeThreads**. They differ from real-time threads in that they do not have access to the heap memory. Without access to the heap, no-heap real-time threads (NHRTs) can continue to run even during a garbage collection cycle, with some restrictions. It follows that without access to the heap the programming model will be different from real-time threads.

Considerations when using NHRTs

- The main reason for using NHRTs is with a task that will not tolerate garbage collection. For example, if your application is time-critical and will not tolerate any interruptions.
- If time is so critical that you are using NHRTs, you should also consider using the ahead-of time (AOT) compiler; that is, use the **-Xnojit** option.
- When you use the **-Xrealtime** option, you automatically use the Metronome Garbage Collector. The benefits of Metronome Garbage Collector might be sufficient for your enterprise, thus reducing the need to code NHRTs.
- NHRT threads run independently of the Garbage Collector because they have a priority higher than that of the Garbage Collector. Java threads can have a priority in the range 1 - 10. If there are NHRTs, the priority of Java threads is reset to 0 regardless of the priority set in your program. The Garbage Collector is automatically set to half a step higher than the highest real-time thread. You set the priority of your NHRTs to be at least one higher than the highest real-time thread. In this way, the NHRTs are independent of the garbage collector.

  Note: NHRTs are not entirely free of garbage collection because the Metronome alarm thread garbage collector runs at the highest priority in the system. This priority ensures that the JVM can be activated to check if the Garbage Collector needs to do anything. The work to run the Metronome alarm thread is very small and would not be expected to affect performance significantly.
- Because NHRTs are restricted to the scoped and immortal memory areas, checks ensure that they are not allocated from the heap. The start method checks and returns an exception (`MemoryAccessError`) if NHRTs are allocated from the heap. NHRTs can access only immortal memory `ImmortalMemory` and scoped memory `ScopedMemory`.
- The semantics of locking are unchanged, so that NHRT threads can be blocked by normal threads if a lock is shared.
- A thread that is using the heap can have its priority boosted on a synchronized method when a NHRT tries to use the same method.
- Use nonblocking queues for communications between NHRTs and heap threads. Otherwise, separate the two types of threads.

Exceptions

- `IllegalAssignmentError`. For example, this error occurs when a reference to scoped memory is attempted to be created in immortal memory.
- `MemoryAccessError`. For example, this error occurs when a NHRT tries to reference heap memory.

Memory and scheduling constraints

The most obvious constraint on `NoHeapRealtimeThreads` (NHRT) is apparent in their name. The JVM prevents NHRTs loading references to objects on the heap onto its operand stack. To do so would throw a `javax.realtime.MemoryAccessError`.

The JVM also guards against references to objects in scoped memory being stored in heap or immortal memory. Although scoped memory is not used exclusively by NHRTs, it is likely to be used if `ImmortalMemory` is inappropriate and memory deallocation is required in an NHRT context.

No-heap real-time threads can overwrite references to heap.

Classloading constraints

Classes are loaded into the same memory areas as the classloader. The default for classloaders is ImmortalMemory.

Applications should be ″warm″ if no-heap real-time threads are to provide the expected response times.

Constraints on Java threads when running with NHRTs

Because system properties are shared within a JVM and any thread can access the system properties, some care is required when using the getProperties and setProperties methods in JVMs in which NHRTs are run. For system properties to be accessible to NHRTs, they must be in immortal memory.

The class java.lang.System provides several methods that allow threads to interact with the system properties. These include:

```
String getProperty(String)
String getProperty(String,String)
Properties getProperties()

String setProperty(String,String)
void setProperties(Properties)
```

The real-time JVM uses an instance of the `com.ibm.realtime.ImmortalProperties` that was created specifically for the real-time JVM object to store all system properties. Use of this instance ensures that any calls to `System.setProperty()` or `System.getProperties.setProperty()` result in the property being stored in immortal memory. No special user code is required in this case but it is important to understand that each time a property is set some immortal memory is consumed.

Calls to `setProperties()` are a little more difficult because the shared Properties object is used to store system properties. When running in a real-time JVM that has NHRTs running, any call to `setProperties` must pass in an instance of an `com.ibm.realtime.ImmortalProperties` (or subclass) that was created in immortal

memory. Use of this instance ensures that all properties that are set following `setProperties` are in immortal memory. It is also important to note that calling `setProperties(null)` results in a new `ImmortalProperties` object being created internally with a default set of properties, all of which consumes additional immortal memory.

Calls to `getProperties()` return either the object that was set or the default properties object, which is an `com.ibm.realtime.ImmortalProperties` object. To maximize compatibility with existing code that calls `getProperties()`, the `ImmortalProperties` object serializes the object and then deserializes in a standard JVM. The default behavior for the serialization of `ImmortalProperties` is to serialize a regular `Properties` object. The default behavior is done because standard JVMs do not have the `ImmortalProperties` object and deserialization would fail. To override this default behavior, `ImmortalProperties` provides the method `enabledReplacement(boolean)`, which, if called with false, disables the default behavior. In this case, serialization serializes the `ImmortalPropeties` object and it is then possible to deserialize this and use the resulting object in a call to `System.setProperties` in a real-time JVM.

Note: Deserialization is done in immortal memory, which might consume an undesirable amount of this limited resource.

Security manager

The security manager set for the system is used by all types of threads within the JVM. For this reason, in a real-time JVM in which NHRTs run, the security manager must be allocated in immortal memory. The real-time JVM ensures that any security manager specified in the command-line options is allocated in immortal memory. The security manager can also be set through calls to System.setSecurityManager(SecurityManager). If the application sets the security manager in this way, it must ensure that the security manager was allocated from Immortal so that NHRTs are able to run correctly.

Note: The exceptions thrown and any objects returned by the security manager must either be in immortal memory, if cached, or be allocated in the current allocation context.

Synchronization

The MonitorControl class and its subclass PriorityInheritance manage synchronization, in particular priority inversion control. These classes allow the setting of a priority inversion control policy either as the default or for specific objects.

The WaitFreeReadQueue, WaitFreeWriteQueue and WaitFreeDequeue classes allow wait-free communication between schedulable objects (especially instances of NoHeapRealtimeThread) and regular Java threads.

The wait-free queue classes provide safe, concurrent access to data shared between instances of NoHeapRealtimeThread and schedulable objects subject to garbage collection delays.

No-heap real-time class safety

There are circumstances where portions of the JSE API cannot necessarily be used in a no-heap context. Restrictions are placed on classes that are shared between heap and no-heap threads. You should be aware of the classes supplied with the JVM that can be safely used.

Sharing objects

Methods that run in no-heap real-time threads throw a javax.realtime.MemoryAccessError whenever they try to load a reference to an object on a heap.

Figure 8 is an example of sort of the code that should be avoided:

```
/**
 * NHRTError1
 *
 * This example is a simple demonstration of an NHRT accessing
 * a heap object reference.
 *
 * The error generated is:
 *
 * Exception in thread "NoHeapRealtimeThread-0" javax.realtime.MemoryAccessError
 *      at NHRT.run(NHRTError1.java:56)
 *      at javax.realtime.RealtimeThread.runImpl(RealtimeThread.java:1754)

 */
import javax.realtime.*;

public class NHRTError1 {
    public static void main(String[] args) {
        NHRTError1 example = new NHRTError1();

        example.run();
    }

    public NHRTError1() {
        message = new String("This on the heap.");
    }

    static public String message; /* The NHRT can access static fields directly - they are always Immortal. */
    static public NHRT myNHRT = null;

    public void run() {
                ImmortalMemory.instance().executeInArea(new Runnable() {
            public void run() {
                NHRTError1.this.myNHRT = new NHRT();
            }
        });

        myNHRT.start();

        try {
            myNHRT.join();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
```

Figure 8. Example of an NHRT accessing a heap object reference

```
    /* A NHRT class */
    class NHRT extends NoHeapRealtimeThread {
        public NHRT() {
            super(null, ImmortalMemory.instance());
        }

        /* Prints the String via the static reference in NHRTError1.message */
        public void run() {
            System.out.println("Message: " + NHRTError1.message);
        }
    }

}
```

Figure 9. Example of an NHRT accessing a heap object reference (continued)

Figure 8 on page 64 produces a javax.realtime.MemoryAccessError:

```
Exception in thread "NoHeapRealtimeThread-0" javax.realtime.MemoryAccessError
        at NHRTError1$NHRT.run(NHRTError1.java:56)
        at javax.realtime.RealtimeThread.runImpl(RealtimeThread.java:1754)
```

If an object is to be accessible to both a no-heap real-time thread and a standard Java thread, the object must be allocated in immortal memory. Similarly, if an object is to be accessible to a no-heap real-time thread and a real-time thread, the object can also be held in a scoped memory area.

In Figure 8 on page 64, the reference to the String ″This on the heap.″ was held within a class variable. This variable is accessible to NHRTs because all classes are allocated within immortal memory. Alternatively, the String could have been passed to the NHRTs constructor.

Most objects contain references to other objects, and so care must be taken when sharing such objects between ordinary threads and NHRTs. A typical example is of a LinkedList allocated in immortal memory, shared between an ordinary thread and a NHRT. If sufficient care is not taken, the standard thread could introduce objects into the LinkedList that are on the heap. Of greater concern is that the data structures that are allocated by the LinkedList to track objects would be allocated on the heap by the ordinary thread, easily causing a MemoryAccessError in the NHRT.

Some classes cannot be safely shared between NHRTs and other threads, regardless of where individual instances of them might be allocated. These are classes that rely on objects stored in class variables, usually for caching purposes. InetAddress is a typical example that caches addresses; if the first thread to call certain methods in InetAddress was running on the heap, the same methods will be unsafe to be called by NHRTs at any point in the future.

Locking on objects with NHRTs

NHRTs should avoid synchronizing with other threads. Consider the following scenario:

- A real-time thread of low priority enters a synchronized block or method, synchronizing on an object.
- An NHRT of high priority is blocked when attempting to synchronize on the same object.
- Priority inheritance causes the real-time thread to temporarily assume the same priority as the NHRT.

- Garbage collection must then run at a higher priority than the NHRT and negates the reason for using the NHRT, which avoids interruption by garbage collection.

It is sometimes unavoidable that NHRTs and other threads synchronize on the same object, but you should minimize the possibility. Be careful when sharing objects to avoid unnecessary synchronization.

Restrictions on safe classes

There are some considerations when an application contains both real-time thread and no-heap real-time thread objects.

- The no-heap real-time thread could suffer MemoryAccessErrors caused by interaction with the real-time thread.
- The no-heap real-time thread might be accidentally delayed by garbage collection caused by the real-time thread.

MemoryAccessErrors caused on no-heap real-time thread

When the two types of thread both invoke methods on the same class, the real-time thread might "pollute" the static variables of the class with objects allocated from the heap. The no-heap real-time thread will receive a MemoryAccessError when trying to access those heap objects. The pollution can also happen on instances of the class. Unfortunately, both problems are quite likely to be seen in typical coding patterns and so it is worth exploring a couple of cases.

If a class is performing a time-consuming operation, it often chooses to cache the result to improve performance of the subsequent operations. The cache is typically a collection such as a HashMap anchored in some static variable in the class. A real-time thread operating in heap context can store a heap object in this collection, which not only adds the object itself but also adds infrastructure objects to the collection; for example, parts of the index. When a no-heap real-time thread later tries to access the collection, even if it is not trying to access the object added by the other thread, it attempts to load the infrastructure objects and hence receive a MemoryAccessError. As class libraries develop and are tuned for performance, these caches become more common.

A class instance can also become polluted by heap objects in a variety of ways. Consider an instance built in immortal memory and thus accessible to both types of thread. If the first use of the object is by a realtime thread in heap context, you might find that a secondary object is stored in a field of the original object. If the secondary object is in heap context, subsequent use by the no-heap real-time thread again shows a MemoryAccessError. These secondary objects might not always be added on first use but after a number of uses and might be designed to improve the performance of heavily used methods.

NoHeap thread delayed by garbage collection

No-heap threads must be assigned priorities that are higher than other threads in order to avoid being delayed by garbage collection.

Additionally, if a class contains any synchronized methods, it is possible that a no-heap real-time thread calling such methods might unintentionally be delayed by garbage collection. This scenario is described in "Locking on objects with NHRTs" on page 65.

If a class contains any synchronized methods (either static or instance methods), it is possible that a no-heap real-time thread calling such methods might unintentionally be delayed by garbage collection. The problem is caused if a real-time thread is accessing a synchronized method (static or instance) at the point where a no-heap real-time thread attempts to invoke another synchronized method that would block waiting for the other thread to complete. If the no-heap real-time thread has a higher priority than the other thread, you would expect priority boosting of that thread to occur. If that thread is then forced to wait for a garbage collection interrupt, there is a potential priority inversion because the garbage collector thread will have a priority higher than the highest priority real-time thread which might not be as high as the no-heap real-time thread that is currently blocked waiting to enter the synchronized method.

The only way to fix such problems is to ensure that no-heap real-time threads never invoke synchronized methods on classes or instances which are shared with other thread types. Unfortunately, it is not always clear from a method signature if a method is synchronized; it might, for example, contain a synchronized block or invoke a synchronized method.

Summary

The NoHeapRealtimeThread class adds a significant amount of complexity to the real-time environment and a significant number of problems can be caused when there is a mixture of thread types operating in an environment. During development of an application, it is very important to carefully design areas in which there is shared use of classes by the different thread types. Of particular importance is the use that these threads make of classes in the SDK. Because of the complexity of analysis, it is impossible to give any guarantee that all the classes provided in the SDK are safe for such shared use. Instead, a small subset of the classes have been verified. Initially, verification has concentrated on the MemoryAccessError aspect and the result is a list of classes that have been analyzed, tested, and modified where necessary to ensure that they can be used by both no-heap and other types of threads.

Safe classes

This section lists the set of classes that are intended to be safely used by both NoHeapRealtimeThread and other thread types. Currently, the main concern is focussed on the MemoryAccessError aspect of safety and the following list details classes that can be used by all three thread types in the same JVM. Note that individual instances of the class might not always be safely shared.

For a class to be safely used by all thread types the usage must initially obey some basic rules. The instance itself must be built in a memory area accessible to the thread intending to access it. If the class has public static fields, the user must avoid storing heap objects in them and the same applies to any public instance fields.

Not all IBM-provided classes are NHRT-safe. The following list describes which classes are safe. If a class is not listed, it is not safe.

java.lang package:*
: All classes in the java.lang package are NHRT safe, except for the following:

Table 11.

Class	Method
java.lang.ProcessBuilder	*
java.lang.Thread	getAllStackTraces()Ljava.util.Map;
java.lang.ThreadGroup	*
java.lang.ThreadLocal	*
java.lang.InheritableThreadLocal	*

java.lang.reflect package:*
All classes in the java.lang.reflect package are NHRT-safe, except for the following:

Table 12.

Class	Method
java.lang.reflect.Proxy.*	*

java.lang.ref package:*
All classes in the java.lang.ref package are NHRT-safe.

java.net package:*
All classes in the java.net package are NHRT-safe, except for the following:

Table 13.

Class	Method
java.net.SocketPermission.*	newPermissionCollection()Ljava.net.SocketPermissionCollection;

java.io package:*
All classes in the java.io package are NHRT-safe, except for the following:

Table 14.

Class	Method
java.io.ExpiringCache	*
java.io.SequenceInputStream	*
java.io.FilePermission	newPermissionCollection()Ljava.io.FilePermissionCollection;
java.io.ObjectInputStream	*
java.io.ObjectOutputStream	*

java.math package:*
All class in the java.math package are NHRT-safe.

A class could be considered NHRT-safe, but might still not be suitable for use in an NHRT because it could have non-deterministic operation. Application developers need to determine the real-time requirements of the classes you use on a case-by-base basis, independent of whether or not a class is NHRT-safe.

The above list of packages is not intended to imply that subpackages are included and thus safe. Hence, for example, the following classes are excluded from this safe list:
- java.lang.management.*
- java.lang.annotation.*
- java.lang.instrument.*

Chapter 9. Troubleshooting in a real-time environment

Dealing with OutOf MemoryError exceptions, memory leaks, and hidden memory allocations

Diagnosing OutOfMemoryErrors

Diagnosing **OutOfMemoryError** exceptions in RTSJ is more complex than in a standard JVM because of the characteristics of the different memory areas in which objects can be allocated. This topic explains how to determine which heap has become full and why.

The characteristics of the different types of heap are described in "Memory management" on page 33. In general, a RTSJ application will require 20% more heap space than a standard Java application.

By default, the JVM will produce the following diagnostics output when an uncaught OutOfMemoryError occurs:

- A snap dump; see "Snap traces" on page 117.
- A Heapdump; see "Using Heapdump" on page 130.
- A Javadump; see "Using Javadump" on page 121

The dump file names are given in the console output:

```
JVMDUMP006I Processing Dump Event "uncaught", detail "java/lang/OutOfMemoryError" - Please Wait.
JVMDUMP007I JVM Requesting Snap Dump using '/home/test/Snap0001.20070115.210154.23027.trc'
JVMDUMP010I Snap Dump written to /home/test/Snap0001.20070115.210154.23027.trc
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump.20070115.210154.23027.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump.20070115.210154.23027.phd
JVMDUMP007I JVM Requesting Java Dump using '/home/test/javacore.20070115.210154.23027.txt'
JVMDUMP010I Java Dump written to /home/test/javacore.20070115.210154.23027.txt
JVMDUMP013I Processed Dump Event "uncaught", detail "java/lang/OutOfMemoryError".
Exception in thread "main" java.lang.OutOfMemoryError
        at java.lang.StringBuilder.ensureCapacityImpl(StringBuilder.java:321)
        at java.lang.StringBuilder.append(StringBuilder.java:207)
        at myOutOfMem.main(myOutOfMem.java:34)
```

The Java backtrace shown on the console output, and also available in the Javadump, indicates where in the Java application the OutOfMemoryError occurred. The next step is to find out which RTSJ memory area is full. The JVM memory management component issues a tracepoint that gives the size, class block address and memory space name of the failing allocation. This tracepoint can be found in the Snap dump:

```
<< lines omitted... >>
13:51:55.994787000*08099a00       j9mm.101  Event       J9AllocateIndexableObject() returning NULL!
  33619996 bytes requested for object of class 0x81312d8 from memory space 'Metronome' id=0x809c5f0
```

The tracepoint ID and data fields may vary from the example above, depending on the type of object being allocated. In this example, the tracepoint shows that the allocation failure occured when the application attempted to allocate a 33.6 MB object of type "class 0x81312d8" in the 'Metronome' heap, memory segment "ID=0x809c5f0".

You can determine which RTSJ memory area is affected by looking at the memory management information in the Javadump:

```
0SECTION       MEMINFO subcomponent dump routine
NULL           ================================
1STHEAPFREE    Bytes of Heap Space Free: 58000
1STHEAPALLOC   Bytes of Heap Space Allocated: 4000000
1STHEAPFREE    Bytes of Immortal Space Free: f66af0
1STHEAPALLOC   Bytes of Immortal Space Allocated: 1000000
NULL
1STSEGTYPE     Internal Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGMENT     08182BA4 083798B0 08379A34 083898B0  01000040 10000
1STSEGMENT     08182A54 083698A8 0837273C 083798A8  01000040 10000
<< lines omitted... >>
NULL
1STSEGTYPE     Object Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGSUBTYPE  Immortal Segment  ID=0809C60C
1STSEGMENT     0809BE00 B284C008 B384C008 B384C008  00001008 1000000
1STSEGSUBTYPE  Heap Segment  ID=0809C5F0
1STSEGMENT     0809BD88 B384D008 B784D008 B784D008  00000009 4000000
<< lines omitted... >>
```

You can determine the type of object being allocated by looking at the classes section of the Javadump:

```
0SECTION       CLASSES subcomponent dump routine
NULL           ================================
<< lines omitted... >>
1CLTEXTCLLOD    ClassLoader loaded classes
2CLTEXTCLLOAD    Loader *System*(0xB2860640)
<< lines omitted... >>
3CLTEXTCLASS      [C(0x081312D8)
<< lines omitted... >>
```

The information in the Javadump confirms that the attempted allocation was for a character array, in the normal heap (ID=0809C5F0) and that the total allocated size of the heap, indicated by the 1STHEAPALLOC line, is 0x400000, or 67 MB.

In this example the failing allocation is very large in relation to the total heap size. If your application is expected to create 33 MB objects, the next step would be to increase the size of the heap, using the **-Xmx** option.

It is more common for the failing allocation to be small in relation to total heap size, with the full heap being caused by previous allocations filling up the heap. In these cases the next step is to use the Heapdump to investigate the amount of memory allocated to existing objects.

The Heapdump is a compressed binary file containing a list of all objects with their object class, size, and references. Analyze the Heapdump using the Memory Dump Diagnostics for Java tool (MDD4J) which is available for download via the IBM Support Assistant (ISA).

Using MDD4J, you can load a Heapdump and locate tree structures of objects that are suspected of consuming large amounts of heap space. The tool provides a variety of views for objects on the heap, Figure 10 on page 71 shows a view ordered by object memory occupancy. It shows that the largest contributor to current heap occupancy is another character array referenced from a String object, in this case of size 16.8 MB. We now have evidence that the application is allocating String objects of very large size in relation to total heap size. The

application in this example is in a loop allocating Strings, where the size of the String is doubling on each iteration.

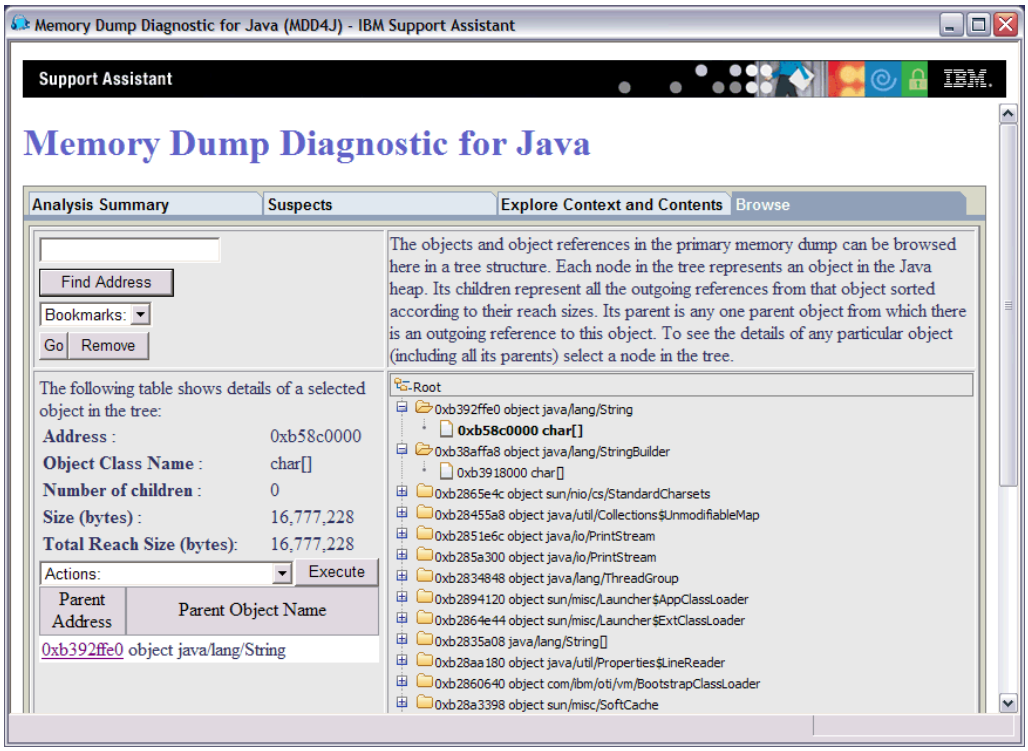

Figure 10. MDD4J showing a large character array in a heap

By default a single Heapdump file containing all objects in all RTSJ memory spaces is produced. Use the command line option **-Xdump:heap:request=multiple** to request a separate Heapdump for each memory space. Using multiple dumps allows you to examine just the set of objects that allocated in a specific memory area. The Heapdumps can be identified by the file name given on the console output:

```
JVMDUMP006I Processing Dump Event "uncaught", detail "java/lang/OutOfMemoryError" - Please Wait.
<< lines omitted... >>
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Default0809DCD8-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Default0809DCD8-0002.phd
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Immortal0809DCF4-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Immortal0809DCF4-0002.phd
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Scope0809DD10-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Scope0809DD10-0002.phd
<< lines omitted... >>
JVMDUMP013I Processed Dump Event "uncaught", detail "java/lang/OutOfMemoryError".
Exception in thread "RTJ Memory Consumer (thread_type=Realtime)" java.lang.OutOfMemoryError
 at tests.com.ibm.jtc.ras.runnable.DepleteMemory.depleteMemory(DepleteMemory.java:57)
<< lines omitted... >>
```

Example OutOfMemoryError in immortal memory space

This example shows how to identify an OutOfMemoryError in the immortal memory space and describes steps to take to prevent the problem.

The Snap dump shows that two small allocation requests have failed in the immortal memory area id=0x809dd1c:

```
16:08:04.876087000 083d4000       j9mm.100  Event       J9AllocateObject() returning NULL!
  16 bytes requested for object of class 0x8110e60 from memory space 'Immortal' id=0x809dd1c
16:08:04.876171000 083d4000       j9mm.100  Event       J9AllocateObject() returning NULL!
  32 bytes requested for object of class 0x81180f0 from memory space 'Immortal' id=0x809dd1c
```

The Java Dump shows that the immortal memory space is full:

```
NULL           ------------------------------------------------------------
0SECTION       MEMINFO subcomponent dump routine
NULL           ================================
1STHEAPFREE    Bytes of Heap Space Free: 3f0c000
1STHEAPALLOC   Bytes of Heap Space Allocated: 4000000
1STHEAPFREE    Bytes of Immortal Space Free: 0
1STHEAPALLOC   Bytes of Immortal Space Allocated: 1000000
<< lines omitted... >>
1STSEGTYPE     Object Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGSUBTYPE  Immortal Segment  ID=0809DD1C
1STSEGMENT     0809D510 B279D008 B379D008 B379D008  00001008 1000000
```

Figure 11 is a screen capture of an MDD4J analysis showing that a very large LinkedList has being allocated, consuming a large proportion of the available memory.

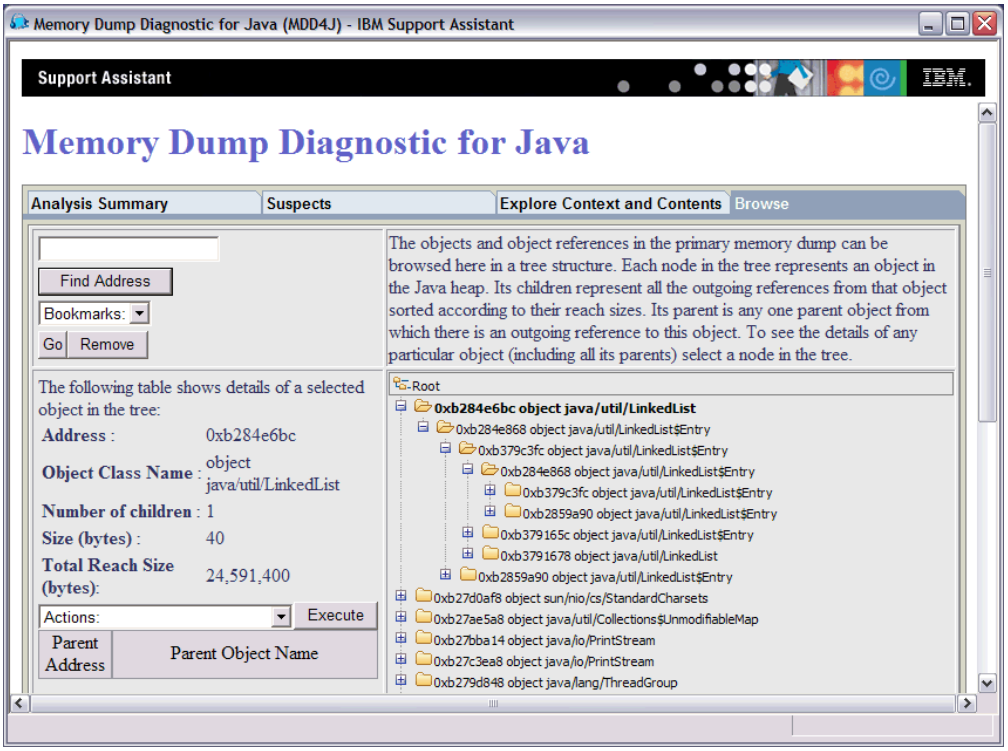

Figure 11. MDD4J showing a large linked list

You should minimize the number of objects allocated within the immortal memory area as objects in the immortal area are not garbage collected. The most common immortal memory usage is class loading, which is a finite activity occurring mostly during JVM and application initialization. Applications with high numbers of loaded classes (or other immortal memory usage) can increase the size of the immortal memory area using the **-Xgc:immortalMemorySize=<size>** parameter. The default size for the immortal memory area is 16 MB.

If increasing the size of the immortal memory area only delays the OutOfMemoryError for immortal memory, investigate the pattern of continued allocation of immortal data, either related to class loading or other application objects.

Example OutOfMemoryError in scoped memory space

This example shows how to identify an OutOfMemoryError in scoped memory space and describes steps to take to prevent the problem.

Use the command-line option **-Xdump:heap:request=multiple** to produce separate dumps for each memory space:

```
VMDUMP006I Processing Dump Event "uncaught", detail "java/lang/OutOfMemoryError" - Please Wait.
JVMDUMP007I JVM Requesting Snap Dump using '/home/test/snap-0001.trc'
JVMDUMP010I Snap Dump written to /home/test/snap-0001.trc
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Default0809DCD8-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Default0809DCD8-0002.phd
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Immortal0809DCF4-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Immortal0809DCF4-0002.phd
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Scope0809DD10-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Scope0809DD10-0002.phd
JVMDUMP007I JVM Requesting Java Dump using '/home/test/javacore-0003.txt'
JVMDUMP010I Java Dump written to /home/test/javacore-0003.txt
JVMDUMP013I Processed Dump Event "uncaught", detail "java/lang/OutOfMemoryError".
Exception in thread "RTJ Memory Consumer (thread_type=Realtime)" java.lang.OutOfMemoryError
   at tests.com.ibm.jtc.ras.runnable.DepleteMemory.depleteMemory(DepleteMemory.java:57)
   at tests.com.ibm.jtc.ras.runnable.DepleteMemory.run(DepleteMemory.java:26)
<< lines omitted... >>
```

The Snap dump shows that two allocation requests have failed in scoped memory area id=0x809dd10:

```
16:14:45.887176823 08480900       j9mm.100  Event       J9AllocateObject() returning NULL!
  16 bytes requested for object of class 0x8110e38 from memory space 'Scoped' id=0x809dd10
16:14:45.887252747 08480900       j9mm.100  Event       J9AllocateObject() returning NULL!
  32 bytes requested for object of class 0x81180c8 from memory space 'Scoped' id=0x809dd10
```

The Javadump shows that for the scoped memory area with id=0x809dd10, the allocated size of the memory area is quite small, only 60 KB; in this case, increase the size of the scoped memory area in the application code.

```
0SECTION       MEMINFO subcomponent dump routine
NULL           ================================
1STHEAPFREE    Bytes of Heap Space Free: 3eb0000
1STHEAPALLOC   Bytes of Heap Space Allocated: 4000000
1STHEAPFREE    Bytes of Immortal Space Free: f47474
1STHEAPALLOC   Bytes of Immortal Space Allocated: 1000000
1STHEAPFREE    Bytes of Scoped Space ID=0809DD10 Free: eb00
1STHEAPALLOC   Bytes of Scoped Space Allocated: eb00
.......
1STSEGTYPE     Object Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGSUBTYPE  Scoped Segment  ID=0809DD10
1STSEGMENT     0809D560 08416350 08424E50 08424E50  00002008 eb00
1STSEGSUBTYPE  Immortal Segment  ID=0809DCF4
1STSEGMENT     0809D4E8 B2857008 B3857008 B3857008  00001008 1000000
```

In the example Javadump the scoped memory area appears to be empty. It appears empty because the Javadump is produced when the OutOfMemoryError reaches the JVM, at which time the scope has been exited and cleaned up. You can produce a Javadump at the point of failure by using the

-Xdump:java:events=throw,filter=java/lang/OutOfMemoryError command-line option. By using this option, the free space in the scoped memory area will be correctly reported.

It is also possible to exhaust the total space available for scoped memory; in this case, increase the size of the scoped memory area using the command-line option **-Xgc:scopedMemoryMaximumSize=<size>**. The default size for the scoped memory area is 8 MB. If the total space available for scoped memory becomes exhausted you will see different messages on the console, for example:

```
Exception in thread "main" java.lang.OutOfMemoryError: Creating (LTMemory) Scoped memory # 0 size=16777216
        at javax.realtime.MemoryArea.create(MemoryArea.java:808)
        at javax.realtime.MemoryArea.create(MemoryArea.java:798)
        at javax.realtime.ScopedMemory.create(ScopedMemory.java:1359)
        at javax.realtime.ScopedMemory.create(ScopedMemory.java:1351)
        at javax.realtime.ScopedMemory.initialize(ScopedMemory.java:1705)
        at javax.realtime.ScopedMemory.<init>(ScopedMemory.java:216)
        at javax.realtime.ScopedMemory.<init>(ScopedMemory.java:164)
```

Diagnosing problems in multiple heaps

You can use the address ranges provided in the Javadump together with the occupancy information in the Heapdump to help analyze OutOfMemoryErrors in multiple RTSJ memory areas.

In this Javadump, the immortal segment ranges from 0xB281C008 to 0xB381C008, and the normal heap segment ranges from 0xB381D008 to 0xB781D008:

```
0SECTION       MEMINFO subcomponent dump routine
NULL           ================================
1STHEAPFREE    Bytes of Heap Space Free: 58000
1STHEAPALLOC   Bytes of Heap Space Allocated: 4000000
1STHEAPFREE    Bytes of Immortal Space Free: b319d8
1STHEAPALLOC   Bytes of Immortal Space Allocated: 1000000
NULL
1STSEGTYPE     Internal Memory
<< lines omitted... >>
1STSEGTYPE     Object Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGSUBTYPE  Immortal Segment  ID=0809C68C
1STSEGMENT     0809BE80 B281C008 B381C008 B381C008  00001008 1000000
1STSEGSUBTYPE  Heap Segment  ID=0809C670
1STSEGMENT     0809BE08 B381D008 B781D008 B781D008  00000009 4000000
NULL
1STSEGTYPE     Class Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGMENT     08158154 083FFD68 083FFEF0 08407D68  00010040 8004
```

The Heapdump is a compressed binary file containing a list of all objects with their object class, size, and references. Analyze the Heapdump using the Memory Dump Diagnostics for Java tool (MDD4J) which is available for download via the IBM Support Assistant (ISA).

You can use the object memory locations listed by MDD4J to determine which memory space an object is in. In Figure 12 on page 75, the first 13 roots objects have addresses in the 0xB28*nnnnn* range, which shows they are in the immortal memory area. The last root object and the Integer objects it contains are in the 0xB61*nnnnn* range, which shows they are in the normal heap.

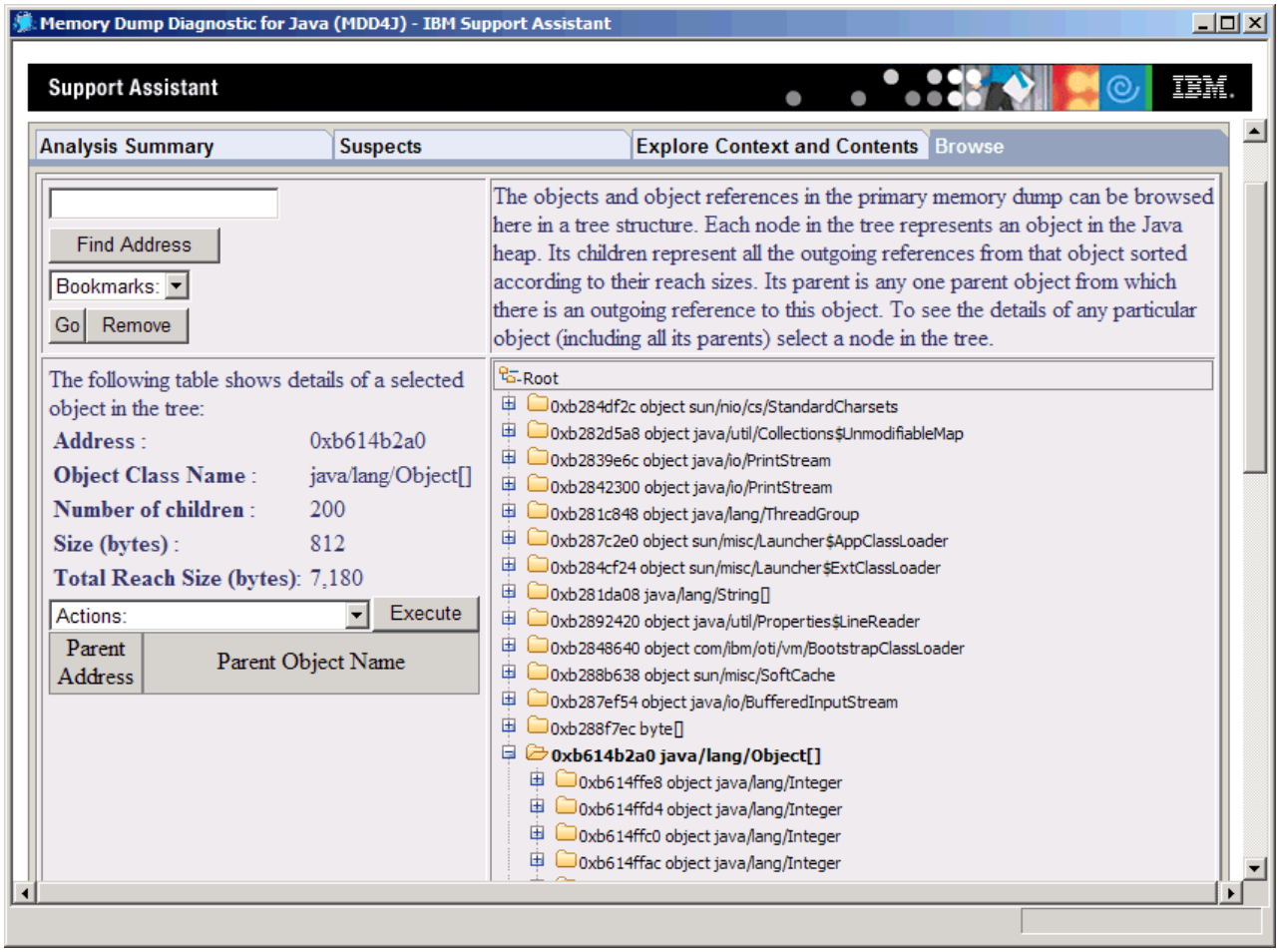

Figure 12. MDD4J showing objects in both the immortal and normal heap memory areas

Avoiding memory leaks

Immortal and scoped memory are not subject to garbage collection. In the case of immortal memory, the memory is only freed when the JVM exits. Scoped memory areas are only freed once their reference counts go to zero. Long running tasks running in these contexts should be written in such a way that after the task has warmed up, no additional memory from the immortal memory area should be allocated.

Loading classes uses a small amount of immortal memory. These classes are not garbage collected in the real-time environment. As such, loading classes that are not required by the application can cause the application to use more immortal memory than necessary.

Objects allocated out of the immortal memory area cannot be garbage collected and, as such, you should be very careful about which objects are allocated from the immortal memory area and should try to minimize usage as much as possible. Object pooling in immortal memory should be considered if the immortal memory area is being used more than occasionally.

Hidden Memory allocation through language features

There is one language feature that should be avoided in a scoped or immortal memory context.

Vararg

Varargs allow a variable number of arguments to be passed to a method. The Java language implements them by passing the **Varargs** in an array to the method, which is expected to treat the **varargs** as an array. The compiler makes calling vararg methods easy by creating and initializing the vararg array for you. Memory can be lost by calling a vararg method in an immortal or scoped memory context.

Varargs should not be used in scoped or immortal memory contexts, but as the array passed to a vararg method can be created explicitly, it can also be reused.

Here is an example showing two equivalent ways of calling a vararg method:

```
public class VarargEx {

    public static void main(String[] args) {
        System.out.println("Sum: "+ sum(1.0, 2.0 , 3.0, 4.0));

        /* ... is the same as ... */

        double array[] = new double[4];

        array[0] = 1.0; array[1] = 2.0; array[2] = 3.0; array[3] = 4.0;
        System.out.println("Sum: " + sum(array));
    }

    static double sum(double... params) {
        double total=0.0;

        for(double num : params) {
            total += num;
        }

        return total;
    }
}
```

Using reflection across memory contexts

If a Constructor object has been built within a scoped memory area, it can only be used within the same scope or an inner scope. Attempting to use that Constructor object in immortal, heap, or an outer scope memory context will fail.

The exception thrown when reflection has occurred across memory contexts will be similar to the following:

```
Exception in thread "NoHeapRealtimeThread-14" javax.realtime.IllegalAssignmentError
   at java.lang.reflect.Constructor$1.<init>(Constructor.java:570)
   at java.lang.reflect.Constructor.acquireConstructorAccessor(Constructor.java:568)
   at java.lang.reflect.Constructor.newInstance(Constructor.java:521)
   at testMain$TestRunnable$1.run(testMain.java:40)
   at javax.realtime.MemoryArea.activateNewArea(MemoryArea.java:597)
   at javax.realtime.MemoryArea.doExecuteInArea(MemoryArea.java:612)
   at javax.realtime.ImmortalMemory.executeInArea(ImmortalMemory.java:77)
   at testMain$TestRunnable.allocate(testMain.java:36)
   at testMain$TestRunnable.run(testMain.java:12)
   at java.lang.Thread.run(Thread.java:875)
   at javax.realtime.ScopedMemory.runEnterLogic(ScopedMemory.java:280)
   at javax.realtime.MemoryArea.enter(MemoryArea.java:159)
```

```
at javax.realtime.ScopedMemory.enterAreaWithCleanup(ScopedMemory.java:194)
at javax.realtime.ScopedMemory.enter(ScopedMemory.java:186)
at javax.realtime.RealtimeThread.runImpl(RealtimeThread.java:1824)
```

It is possible to work around this restriction by using the Constructor in the same scope as it was allocated.

Chapter 10. Problem determination

This section is intended to help you find the kind of fault you have and from there to do one or more of the following tasks:

- Fix the problem
- Find a good workaround
- Collect the necessary data with which to generate a bug report to IBM

A couple of JVM areas do not fit neatly into the platform model, and these have their own sections:

- "ORB problem determination" on page 96
- "NLS problem determination" on page 109

The sections in this part are:

- "First steps in problem determination"
- "Linux problem determination" on page 81
- "ORB problem determination" on page 96
- "NLS problem determination" on page 109

First steps in problem determination

Ask these questions before going any further.

Have you changed anything recently?
If you have changed, added, or removed software of hardware just before the problem occurred, back out the change and see if the problem persists.

What else is running on the machine?
If you have other software, including a firewall, try switching it off to see if the problem persists.

Is the problem reproducible on the same machine?
Knowing that this defect occurs every time the described steps are taken, is one of the most helpful things you can know about it and tends to indicate a straightforward programming error. If, however, it occurs at alternate times, or at one time in ten or a hundred, thread interaction and timing problems in general would be much more likely.

Is the problem reproducible on another machine?
A problem that is not evident on another machine could help you find the cause. A difference in hardware could make the problem disappear; for example, the number of processors. Also, differences in the operating system and application software installed might make a difference to the JVM. For example, the visibility of a race condition in the JVM or a user Java application might be influenced by the speed at which certain operations are performed by the system.

Does the problem appear on multiple platforms?
If the problem appears only on one platform, it could be related to a platform-specific part of the JVM or native code used within a user application. If the problem occurs on multiple platforms, the problem could be related to the user Java application or a cross-platform part of the JVM; for example, Java Swing API. Some problems might be evident only

on particular hardware; for example, Intel32. A problem on particular hardware could possibly indicate a JIT problem.

Can you reproduce the problem with the latest Service Refresh?
The problem might also have been fixed in a recent service refresh. Make sure you are using the latest service refresh for your environment. Check for latest details on http://www.ibm.com/developerWorks.

Are you using a supported Operating System (OS) with the latest patches installed?
It is important to use RHEL 4 Realtime Linux and to have the latest patches for operating system components. For example, upgrading system libraries can solve problems. Moreover, later versions of system software can provide a richer set of diagnostic information. For more information, see "Linux problem determination" on page 81 and check for latest details on http://www.ibm.com/developerWorks.

Does turning off the JIT help?
If turning off the JIT prevents the problem, there might be a problem with the JIT. This can also indicate a race condition within the user Java application which surfaces only in certain conditions. If the problem is intermittent, reducing the JIT compilation threshold to 0 might help reproduce the problem more consistently. (See "Problem determination for compilers" on page 154.)

Have you tried reinstalling the JVM or other software and rebuilding relevant application files?
Some problems occur from a damaged or invalid installation of the JVM or other software. It is also possible that an application could have inconsistent versions of binary files or packages. Inconsistency is particularly likely in a development or testing environment and could potentially be solved by getting a completely fresh build or installation.

Is the problem particular to a multiprocessor (or SMP) platform? If you are working on a multiprocessor platform, does the problem still exist on a uniprocessor platform?
This information is valuable to IBM Service.

Have you installed the latest patches for other software that interacts with the JVM? For example, the IBM WebSphere Application Server and DB2®.
The problem could be related to configuration of the JVM in a larger environment and might have been solved already in a fixpack. Is the problem reproducible when the latest patches have been installed?

Have you enabled core dumps?
Core dumps are essential to enable IBM Service to debug a problem. Although core dumps are enabled by default for the Java process (see "Using dump and trace agents" on page 111 for details), operating system settings might also need to be in place to allow the dump to be generated and to ensure that it is complete. Details of the required Linux settings are contained in "Linux problem determination" on page 81.

Linux problem determination

This section describes problem determination on Linux for WebSphere Real Time.

Setting up and checking your Linux environment

Linux operating systems undergo a large number of patches and updates.

Working directory

The current working directory of the JVM process is the default location for the generation of core files, Java dumps, heap dumps, and the JVM trace outputs, including Application Trace and Method trace. Enough free disk space must be available for this directory. Also, the JVM must have write permission.

Linux core files

When a crash occurs, the most important diagnostic data to obtain is the process core file. To ensure that this file is generated for the JVM on Linux, you make a number of settings. Most JVM settings are defaults, but operating system settings must also be correct, and these vary by distribution and Linux version. The following settings must be in place.

Operating system settings

To obtain full core files, set the following `ulimit` options:

```
ulimit -c unlimited   turn on corefiles with unlimited size
ulimit -n unlimited   allows an unlimited number of open file descriptors
ulimit -u unlimited   sets the user memory limit to unlimited
ulimit -f unlimited   sets the file limit to unlimited
```

The commands above modify both hard and soft limits. To modify just the "soft" limits, the same `ulimit` commands can be run using the additional **-S** flag. To modify just the "hard" limits use the **-H** flag.

Note: The hard limit can only be modified downwards and the soft limit cannot exceed the hard limit.

The current `ulimit` settings can be displayed using:

```
ulimit -a
```

The values shown are the "soft" limits for the process, to display the hard limits use the **-H** flag. From Java Version 5 and WebSphere Real Time, the `ulimit -c` value for the soft limit is ignored and the hard limit value is used to help ensure generation of the core file. You can disable core file generation by using the **-Xdump:system:none** command-line option.

Java Virtual machine settings

To generate core files when a crash occurs, check that the JVM is set to do so. Run **java -Xrealtime -Xdump:what**, which should produce the following:

```
dumpFn=doSystemDump
events=gpf+abort
filter=
label=/u/cbailey/sdk/jre/bin/core.%Y%m%d.%H%M%S.%pid.dmp
range=1..0
priority=999
request=serial
opts=
```

The values above are the default settings. At least `events=gpf` must be set to generate a core file when a crash occurs. You can change and set options with the command-line option **-Xdump:system[:name1=value1,name2=value2 ...]**

Available disk space

The disk space must be sufficient for the core file to be written. The JVM allows the core file to be written to any directory that is specified in the `label` option. For example:

```
-Xdump:system:label=/u/cbailey/sdk/jre/bin/core.%Y%m%d.%H%M%S.%pid.dmp
```

To write the core file to this location, you need up to 4 GB for a 32-bit process, more for a 64-bit process. You also need the correct permissions for the Java process to write to that location.

Using CPU Time limits to control runaway tasks

Since real time threads run at high priorities and with FIFO scheduling, failing applications (typically with tight CPU-bound loops) can cause a system to become unresponsive. In a development environment it can be useful to ensure runaway tasks are killed by limiting the amount of CPU that tasks may consume. See "Linux core files" on page 81 for a discussion on soft and hard limit settings.

The command `ulimit -t` lists the current timeout value in CPU seconds. This value can be reduced with either soft, for example, `ulimit -St 900` to set the soft timeout to 15 minutes or hard values to stop runaway tasks.

General debugging techniques

This section provides a guide to the JVM-provided diagnostic tools and Linux commands that can be useful when you are diagnosing problems that occur with the Linux JVM.

Table 15.

Action	Reference
Starting Javadumps	See "Using Javadump" on page 121.
Starting Heapumps	See "Using Heapdump" on page 130.

Using the dump extractor on Linux

When a core dump occurs, the following items are required so that you can analyze the core file: The core file, a copy of the Java executable that was running the process, and copies of all the libraries that were in use when the process core dumped.

When a core file is generated, run the jextract utility against the core file:

```
jextract <core file name>
```

or for WebSphere Real Time enter:

```
jextract -Xrealtime <core file name>
```

to generate a file called dumpfilename.zip in the current directory. dumpfilename.zip is a compressed file containing the required files. Running jextract against the core file also allows for the subsequent use of the Dump Viewer.

Using core dumps

The commands **objdump**, **nm**, **jextract**, and **jdmpview** are used to investigate and display information about core dumps. If a crash occurs and a corefile is produced, these commands help you analyze the file.

objdump
: Use this command to disassemble shared objects and libraries. After you have discovered which library or object has caused the problem, use **objdump** to locate the method in which the problem originates. To invoke objdump, enter: `objdump <option> <filename>`

nm
: This command lists symbol names from object files. These symbol names can be either functions, global variables, or static variables. For each symbol, the value, symbol type, and symbol name are displayed. Lower case symbol types mean the symbol is local, while upper case means the symbol is global or external. To use this tool, type: `nm <option> <filename>`.

jextract
: Use this command to obtain platform specific information from a system dump. See "Dump extractor: jextract" on page 140 for more information.

jmpdump -Xrealtime
: Use this command to examine the contents of system dumps produced from the JVM. See "Dump viewer: jdmpview -Xrealtime" on page 141 for more information.

You can see a complete list of options by typing `objdump -H`. The **-d** option disassembles contents of executable sections

Run these commands on the same machine as the one that produced the core files to get the most accurate symbolic information available. This output (together with the core file, if small enough) is used by the IBM support team for Java to diagnose a problem.

Using system logs

The kernel provides useful environment information. These commands can be used to view this information.

- `ps -eLo pid,tid,policy,rtprio,command`
- `top`
- `vmstat`

The ps command displays process status. Use it to gather information about native threads. In this example. the ps command gives information on the process id, thread id, scheduling policy, real-time thread priority, and the command associated with the process.

The **top** command displays the most CPU- or memory-intensive processes in real time. It provides an interactive interface for manipulation of processes and allows sorting by different criteria, such as CPU usage or memory usage. The display is updated every five seconds by default, although this can be changed by using the **s** (interactive) command. The **top** command displays several fields of information for each process. The process field shows the total number of processes that are running, but breaks this down into tasks that are running, sleeping, stopped, or undead. In addition to displaying PID, PPID, and UID, the **top** command displays information on memory usage and swap space. The mem field shows statistics on memory usage, including available memory, free memory, used memory, shared memory, and memory used for buffers. The Swap field shows total swap space, available swap space, and used swap space.

The **vmstat** command reports virtual memory statistics. It is useful to perform a general health check on your system, although, because it reports on the system as a whole, commands such as ps and top can be used afterwards to gain more specific information about your process's operation. When you use it for the first time during a session, the information is reported as averages since the last reboot. However, further usage will display reports that are based on a sampling period that you can specify as an option. `Vmstat 3 4` will display values every 3 seconds for a count of 4 times. It might be useful to start vmstat before the application, have it direct its output to a file and later study the statistics as the application started and ran. The basic output from this command appears in six sections; processes, memory, swap, io, system, and cpu.

The **processes** section shows how many processes are awaiting run time, blocked, or swapped out.

The **memory** section shows the amount of memory (in kilobytes) swapped, free, buffered, and cached. If the free memory is going down during certain stages of your applications execution, there might be a memory leak.

The **swap** section shows the kilobytes per second of memory swapped in from and swapped out to disk. Memory is swapped out to disk if RAM is not sufficient to store it all. Large values here can be a hint that not enough RAM is available (although it is normal to get swapping when the application first starts).

The **io** section shows the number of blocks per second of memory sent to and received from block devices.

The **system** section displays the interrupts and the context switches per second. There is overhead associated with each context switch so a high value for this may mean that the program does not scale well.

The **cpu** section shows a break down of processor time between user time, system time, and idle time. The idle time figure shows how busy a processor is, with a low value indicating that the processor is very busy. You can use this knowledge to help you understand which areas of your program are using the CPU the most.

Linux debugging commands

These commands can be used for debugging.

ps:

On Linux, Java threads are implemented as system threads and might be visible in the process table, depending on the Linux distribution. Running the ps command gives you a snapshot of the current processes. The ps command gets its information from the /proc filesystem.

Here is an example of using ps:

```
ps -eLo pid,tid,policy,rtprio,command

 PID   TID POL RTPRIO COMMAND
9723  9723  TS  -     java -Xrealtime app
9723  9724  FF  89    java -Xrealtime app
9723  9729  TS  -     java -Xrealtime app
9723  9730  TS  -     java -Xrealtime app
9723  9731  TS  -     java -Xrealtime app
9723  9732  FF  12    java -Xrealtime app
9723  9733  TS  -     java -Xrealtime app
9723  9734  FF  89    java -Xrealtime app
9723  9821  FF  43    java -Xrealtime app
9723  9822  FF  43    java -Xrealtime app
9723  9823  FF  83    java -Xrealtime app
9723  9824  FF  83    java -Xrealtime app
9723  9825  FF  83    java -Xrealtime app
9723  9826  FF  83    java -Xrealtime app
9723  9827  FF  83    java -Xrealtime app
9723  9828  FF  83    java -Xrealtime app
9723  9829  FF  83    java -Xrealtime app
9723  9830  FF  85    java -Xrealtime app
9723  9831  FF  85    java -Xrealtime app
9723  9832  FF  87    java -Xrealtime app
9723  9833  FF  83    java -Xrealtime app
9723  9834  FF  85    java -Xrealtime app
9723  9888  TS  -     java -Xrealtime app
```

e Selects all processes.

L Shows threads.

o Provides a pre-defined format of columns to display.

Note: The columns specified are the process id, thread id, scheduling policy, real-time thread priority, and the command associated with the process. This is useful for understanding what threads in your application as well as the virtual machine are running at a given time.

Tracing:

Tracing is a technique that presents details of the execution of your program. If you are able to follow the path of execution, you will gain a better insight into how your program runs and interacts with its environment. Also, you will be able to pinpoint locations where your program starts to deviate from its expected behavior.

Three tracing tools on Linux are **strace**, **ltrace**, and **mtrace**. The command `man strace` displays a full set of available options.

strace The strace tool traces system calls. You can either use it on a process that is already active, or start it with a new process. strace records the system calls made by a program and the signals received by a process. For each system call, the name, arguments, and return value are used. strace allows you to trace a program without requiring the source (no recompilation is required). If you use it with the **-f** option, it will trace child processes that

have been created as a result of a forked system call. strace is often used to investigate plug-in problems or to try to understand why programs do not start properly.

ltrace The ltrace tool is distribution-dependent. It is very similar to strace. This tool intercepts and records the dynamic library calls as called by the executing process. strace does the same for the signals received by the executing process.

mtrace
mtrace is included in the GNU toolset. It installs special handlers for malloc, realloc, and free, and enables all uses of these functions to be traced and recorded to a file. This tracing decreases program efficiency and should not be enabled during normal use. To use mtrace, set **IBM_MALLOCTRACE** to 1, and set **MALLOC_TRACE** to point to a valid file where the tracing information will be stored. You must have write access to this file.

gdb:

The GNU debugger (gdb) allows you to examine the internals of another program while the program executes or retrospectively to see what a program was doing at the moment that it crashed.

The gdb allows you to examine and control the execution of code and is very useful for evaluating the causes of crashes or general incorrect behavior. gdb does not handle Java processes, so it is of limited use on a pure Java program. It is useful for debugging native libraries and the JVM itself.

You can run gdb in three ways:

Starting a program
Normally the command: `gdb <application>` is used to start a program under the control of gdb. However, because of the way that Java is launched, you must invoke gdb by setting an environment variable and then calling Java:

```
export IBM_JVM_DEBUG_PROG=gdb
java
```

Then you receive a gdb prompt, and you supply the run command and the Java arguments:

```
r<java_arguments>
```

Attaching to a running program
If a Java program is already running, you can control it under gdb. The process id of the running program is required, and then gdb is started with the Java executable as the first argument and the pid as the second argument:

```
gdb <Java Executable> <PID>
```

When gdb is attached to a running program, this program is halted and its position within the code is displayed for the viewer. The program is then under the control of gdb and you can start to issue commands to set and view the variables and generally control the execution of the code.

Running on a corefile
A corefile is normally produced when a program crashes. gdb can be run on this corefile. The corefile contains the state of the program when the

crash occurred. Use gdb to examine the values of all the variables and registers leading up to a crash. With this information, you should be able to discover what caused the crash. To debug a corefile, invoke gdb with the Java executable as the first argument and the corefile name as the second argument:

```
gdb <Java Executable> <corefile>
```

When you run gdb against a corefile, it will initially show information such as the termination signal the program received, the function that was executing at the time, and even the line of code that generated the fault.

When a program comes under the control of gdb, a welcome message is displayed followed by a promdt (gdb). The program is now waiting for your input and will continue in whichever way you choose.

There are a number of ways of controlling execution and examination of the code. Breakpoints can be set for a particular line or function using the command:

breakpoint linenumber
or
breakpoint functionName

After you have set a breakpoint, use the **continue** command to allow the program to execute until it hits a breakpoint.

Set breakpoints using conditionals so that the program will halt only when the specified condition is reached. For example, using **breakpoint 39 if var = = value** causes the program to halt on line 39 only if the variable is equal to the specified value.

If you want to know *where* as well as *when* a variable became a certain value you can use a watchpoint. Set the watchpoint when the variable in question is in scope. After doing so, you will be alerted whenever this variable attains the specified value. The syntax of the command is: `watch var = = value`.

To see which breakpoints and watchpoints are set, use the **info** command:

info break
info watch

When gdb reaches a breakpoint or watchpoint, it prints out the line of code it is next set to execute. Note that setting a breakpoint on line 8 will cause the program to halt after completing execution of line 7 but before execution of line 8. As well as breakpoints and watchpoints, the program also halts when it receives certain system signals. By using the following commands, you can stop the debugging tool halting every time it receives these system signals:

handle sig32 pass nostop noprint
handle sigusr2 pass nostop noprint

When the correct position of the code has been reached, there are a number of ways to examine the code. The most useful is **backtrace** (abbreviated to **bt**), which shows the call stack. The call stack is the collection of function frames, where each function frame contains information such as function parameters and local variables. These function frames are placed on the call stack in the order that they are executed (the most recently called function appears at the top of the call stack),

so you can follow the trail of execution of a program by examining the call stack. When the call stack is displayed, it shows a frame number to the very left, followed by the address of the calling function, followed by the function name and the source file for the function. For example:

#6 0x804c4d8 in myFunction () at myApplication.c

To view more detailed information about a function frame, use the **frame** command along with a parameter specifying the frame number. After you have selected a frame, you can display its variables using the command `print var`.

Use the **print** command to change the value of a variable; for example, `print var = newValue`.

The **info locals** command displays the values of all local variables in the selected function.

To follow the exact sequence of execution of your program, use the **step** and **next** commands. Both commands take an optional parameter specifying the number of lines to execute, but while **next** treats function calls as a single line of execution, **step** steps through each line of the called function.

When you have finished debugging your code, the **run** command causes the program to be restarted. The **quit** command is used to exit gdb.

Other useful commands are:

dtype Prints datatype of variable.

info share
Prints the names of the shared libraries that are currently loaded.

info functions
Prints all the function prototypes.

list Shows the 10 lines of source code around the current line.

help The **help** command displays a list of subjects, each of which can have the help command invoked on it, to display detailed help on that topic.

Diagnosing crashes

Many approaches are possible when you are trying to determine the cause of a crash. The process normally involves isolating the problem by checking the system setup and trying various diagnostic options.

Checking the system environment

The system might have been in a state that has caused the JVM to crash. For example, this could be a resource shortage (such as memory or disk) or a stability problem. Check the Javadump file, which contains various system information (as described in “Using Javadump” on page 121). The Javadump file tells you how to find disk and memory resource information. The system logs can give indications of system problems.

Gathering process information

It is useful to find out what exactly was happening leading up to the crash.

Use gdb and the bt command to display the stack trace of the failing thread and show what was running up to the point of the crash. This could be:

- JNI native code.
- JIT compiled code. If you have a problem with the JIT, try running with JIT off by setting the **-Xint** option.
- JVM code.

Other tracing methods:

- **ltrace**
- **strace**
- **mtrace** - can be used to track memory calls and determine possible corruption
- RAS trace, described in *Using the Reliability, Availability, and Servicability Interface* in the Diagnostics Guide.

Finding out about the Java environment

Use the Javadump to determine what each thread was doing and which Java methods were being executed. Match function addresses against library addresses to determine the source of code executing at various points.

Use the **verbose:gc** option to look at the state of the Java heap, Immortal and Scoped Memory areas and determine if:

- There was a shortage of memory in one of the memory areas and if this could have caused the crash.
- The crash occurred during garbage collection, indicating a possible garbage collection fault. See “Garbage Collector diagnostics” on page 158.
- The crash occurred after garbage collection, indicating a possible memory corruption.

For more information about the Garbage Collector, see *Understanding the Garbage Collector* in the Diagnostics Guide.

Debugging hangs

A hang is caused by a wait (consuming no CPU) or a loop (consuming CPU). A wait could either be caused by some timing error leading to a missed notification or by two threads deadlocking on resources. In this case one thread holds resource A and wants resource B whilst another thread holds resource B and wants resource A. A loop on the other hand is caused by a thread failing to exit a loop in a timely manner. This may be because it calculated the wrong limit value or it may miss some flag setting that was intended to terminate the loop. This problem may only appear on multi processor machines and if so can usually be traced to a failure to make the flag volatile or access it whilst holding an appropriate monitor.

For an explanation of deadlocks and diagnosing them using the Javadump tool, see “Locks, monitors, and deadlocks (LOCKS)” on page 126.

The following approaches are useful to resolve wait of loops:

- Monitoring process and system state (as described in “Gathering information for Linux” on page 93).
- Java Dumps give monitor and lock information.

Debugging memory leaks

If dynamically allocated objects are not freed at the end of their lifetime, memory leaks can occur. When objects that should have had their memory released are still holding memory and more objects are being created, the system eventually runs out of memory.

The **mtrace** tool from GNU is available for tracking memory calls. This tool enables you to trace memory calls such as malloc and realloc so that you can detect and locate memory leaks.

For more details about analyzing the Java Heap, see "Using Heapdump" on page 130.

Debugging performance problems

Locating the causes of poor performance is often difficult. Although many factors can affect performance, the overall effect is generally perceived as poor response or slow execution of your program.

Correcting one performance problem might cause more problems in another area. By finding and correcting a bottleneck in one place you might only shift the cause of poor performance to other areas. To improve performance, experiment with tuning different parameters, monitoring the effect, and retuning until you are satisfied that your system is performing acceptably

Finding the bottleneck

Given that any performance problem could be caused by any one of several factors, you must look at several areas to eliminate each one.

Determine which resource is constraining the system:

- CPU
- Memory
- Input/Output (I/O)

Several tools (see "Application profiling" on page 92 for examples of suitable tools) are available that enable you to measure system components and establish how they are performing and under what kind of workload.

The aspects of the system that you are most interested in measuring are CPU usage and memory usage. If you can prove that the CPU is not powerful enough to handle the workload, any amount of tuning makes not much difference to overall performance. Nothing less than a CPU upgrade might be required. Similarly, if a program is running in an environment in which it does not have enough memory, an increase in the memory is going to make a much bigger change to performance than any amount of tuning does.

CPU usage

You might typically experience Java processes consuming 100% of processor time when a process reaches its resource limits. Ensure that **ulimit** settings are appropriate to the application requirement.

Note: See "Linux core files" on page 81 for more information about **ulimit**.

The /proc file system provides information about all the processes that are running on your system, including the Linux kernel. See man proc from a Linux shell for official Linux documentation on the /proc file system.

The **top** command provides real-time information about your system processes. The **top** command is useful for getting an overview of the system load. It clearly displays which processes are using the most resources. Having identified the processes that are probably causing a degraded performance, you can take further steps to improve the overall efficiency of your program. More information is provided about the **top** command in "Using system logs" on page 83.

Memory usage

If a system is performing poorly because of lack of memory resources, it is memory bound. By viewing the contents of /proc/meminfo, (for example, `cat /proc/meminfo` from a Linux shell); you can view your memory resources and see how they are being used. /proc/swap gives information on your swap file.

Swap space is used as an extension of the systems virtual memory. Therefore, not having enough memory or swap space causes performance problems. A general guideline is that swap space should be at least twice as large as the physical memory.

A swap space can be either a file or disk partition. A disk partition offers better performance than a file does. **fdisk** and **cfdisk** are the commands that you use to create another swap partition. It is a good idea to create swap partitions on different disk drives because this distributes the I/O activities and so reduces the chance of further bottlenecks.

vmstat is a tool that enables you to discover where performance problems might be caused. For example, if you see that high swap rates are occurring, it is likely that you do not have enough physical or swap space. The **free** command displays your memory configuration; **swapon -s** displays your swap device configuration. A high swap rate (for example, many page faults) means that it is quite likely that you need to increase your physical memory. More details on how to use VMstat are provided in "Using system logs" on page 83.

Network problems

Another area that often affects performance is the network. Obviously, the more you know about the behavior of your program, the easier it is for you to decide whether this is a likely source of performance bottleneck.

If you think that your program is likely to be network I/O bound, netstat is a useful tool. In addition to providing information about network routes, netstat gives a list of active sockets for each network protocol and can give overall statistics, such as the number of packets that are received and sent. Using netstat, you can see how many sockets are in a CLOSE_WAIT or ESTABLISHED state and you can tune the respective TCP/IP parameters accordingly for better performance of the system. For example, tuning /proc/sys/net/ipv4/tcp_keepalive_time will reduce the time for socket waits in TIMED_WAIT state before closing a socket. If you are tuning /proc/sys/net file system, the effect will be on all the applications running on the system. However, to make a change to an individual socket or connection, you have to use Java Socket API calls (on the respective socket object). Use **netstat -p** (or the **lsof** command) to find the right PID of a particular socket connection and its stack trace from a javacore file taken with the **kill -3 <pid>** command.

You can also use IBM's RAS trace, **-Xtrace:print=net**, to trace out network-related activity within the JVM. This technique is helpful when socket-related Java thread hangs are seen. Correlating output from **netstat -p**, **lsof**, JVM net trace, and **ps -efH** can help you to diagnose the network-related problems.

Providing summary statistics that are related to your network is useful for investigating programs that might be underperforming because of TCP/IP problems. The more you understand your hardware capacity, the easier it is for you to tune with confidence the parameters of particular system components that will improve the overall performance of your application. You can also determine whether only system tuning and tweaking will noticeably improve performance, or whether actual upgrades are required.

Sizing memory areas

The JVM can be tuned by varying the sizes of the heap, immortal and scoped memory. Poorly chosen sizes can result in significant performance problems as the Garbage Collector has to work harder to stay ahead of utilization.

For more information about varying the size of the memory areas see "Troubleshooting the Metronome Garbage Collector" on page 23.

JIT compilation and performance

The JIT is another area that can affect the performance of your program. When deciding whether or not to use JIT compilation, you should consider the implications to your real-time behavior.

If you require predictable behavior but also need better performance then you should consider using ahead-of-time (AOT) compilation. For further information see Chapter 3, "Using the ahead-of-time compiler and WebSphere Real Time," on page 15 with WebSphere Real Time.

Application profiling

You can learn a lot about your Java application by using the hprof profiling agent. Statistics about CPU and memory usage are presented along with many other options.

The hprof tool is discussed in detail in Diagnostics Guide. **-Xrunhprof:help** gives you a list of suboptions that you can use with hprof.

The Performance Inspector™ package contains a suite of performance analysis tools for Linux. You can use tools to help identify performance problems in your application as well as to understand how your application interacts with the Linux kernel. See http://perfinsp.sourceforge.net/ for details.

Gathering information for Linux

When a problem occurs, the more information known about the state of the system environment, the easier it is to reach a diagnosis of the problem. A large set of information can be collected, although only some of it will be relevant for particular problems. The following sections tell you the data to collect to help the IBM service team for Java solve the problem.

Collecting core files

Collect corefiles to help diagnose many types of problem. Process the corefile with jextract. The resultant jar file is useful for service. See "Dump extractor: jextract" on page 140 for more information.

Producing Javadumps

In some conditions (a crash, for example), a Javadump is produced, usually in the current directory. In others, for example, a hang), you might have to prompt the JVM for this by sending the JVM a `SIGQUIT (kill -3 <PID>)` signal. This is discussed in more detail in "Using Javadump" on page 121.

Using system dumps

The kernel logs system messages and warnings. The system log is located in the /var/log/messages file. Use it to observe the actions that led to a particular problem or event. The system log can also help you determine the state of a system. Other system logs are in the /var/log directory.

Determining the operating environment

The following commands can be useful to determine the operating environment of a process at various stages of its lifecycle:

uname -a
Provides operating system and hardware information.

df Displays free disk space on a system.

free Displays memory use information.

ps -eLo pid,tid,policy,rtprio,command
Gives a full process list.

lsof Lists open file handles.

top Displays process information (such as processor, memory, states) sorted by default by processor usage.

vmstat
Provides general memory and paging information.

In general, the **uname**, **df**, and **free** output is the most useful. The other commands may be run before and after a crash or during a hang to determine the state of a process and to provide useful diagnostic information.

Sending information to Java support

When you have collected the output of the commands listed in the previous section, put that output into files. Compress the files (which could be very large) before sending them to Java Support. You should compress the files at a very high ratio.

The following command builds an archive from files {file1,..,fileN} and compresses them to a file whose name has the format filename.tgz:

```
tar cjf filename.tgz file1 file2...filen
```

Collecting additional diagnostic data

Depending on the type of problem, the following data can also help you diagnose problems. The information available depends on the way in which Java is invoked and also the system environment. You will probably have to change the setup and then restart Java to reproduce the problem with these debugging aids switched on.

proc file system

The /proc file system gives direct access to kernel level information. The /proc/N directory contains detailed diagnostic information about the process with PID (process id) N, where N is the id of the process.

The command `cat /proc/N/maps` lists memory segments (including native heap) for a given process.

strace, ltrace, and mtrace

Use the commands **strace**, **ltrace**, and **mtrace** to collect further diagnostic data. See "Tracing" on page 85.

Heapdumps

The JVM can generate a Heapdump at the request of the user (for example by calling com.ibm.jvm.Dump.HeapDump() from within the application) or (by default) when the JVM terminates because of an OutOfMemoryError. You can specify finer control of the timing of a Heapdump with the **-Xdump:heap** option. For example, you could request a heapdump after a certain number of full garbage collections have occurred. The default heapdump format (phd files) is not human-readable and you process it using available tools such as Heaproots. See "Using Heapdump" on page 130 for more details.

Snap trace

Under default conditions, a running JVM collects a small amount of trace data in a special wraparound buffer. This data is dumped to file when the JVM terminates unexpectedly or an OutOfMemoryError occurs. You can use the **-Xdump:snap** option to vary the events that cause a snap trace to be produced. The snap trace is in normal trace file format and requires the use of the supplied standard trace formatter so that you can read it. See *Tracing Java applications and the JVM* in the Diagnostics Guide for more details on the contents and control of snap traces.

Known limitations on Linux

Linux has been under rapid development and there have been various issues with the interaction of the JVM and the operating system, particularly in the area of threads.

Floating stack limitations

If you are running without floating stacks, regardless of what is set for **-Xss<value>**, a minimum native stack size of 256 KB for each thread is provided. On a floating stack Linux system, the **-Xss<value>** values are used. Thus, if you are migrating from a non-floating stack Linux system, ensure that any **-Xss<value>** values are large enough and are not relying on a minimum of 256 KB.

Font limitations

When you are installing WebSphere Real Time, to allow the font server to find the Java TrueType fonts, run (on Linux IA32, for example):

```
/usr/sbin/chkfontpath --add /opt/ibm/java2-i386-50/jre/lib/fonts
```

You must do this at install time and you must be logged on as "root" to run the command. For more detailed font issues, see the *Linux SDK and Runtime Environment User Guide*.

ORB problem determination

One of your first tasks when debugging an ORB problem is to determine whether the problem is in the client-side or in the server-side of the distributed application. Think of a typical RMI-IIOP session as a simple, synchronous communication between a client that is requesting access to an object, and a server that is providing it.

During this communication, a problem might occur in the execution of one of the following steps:

1. The client writes and sends a request to the server.
2. The server receives and reads the request.
3. The server executes the task in the request.
4. The server writes and sends a reply back.
5. The client receives and reads the reply.

It is not always easy to identify where the problem occurred. Often, the information that the application returns, in the form of stack traces or error messages, is not enough for you to make a decision. Also, because the client and server communicate through their ORBs, it is likely that if a problem occurs, both sides will record an exception or unusual behavior.

This section describes all the clues that you can use to find the source of the ORB problem. It also describes a few common problems that occur more frequently. The topics are:

Identifying an ORB problem

A background of the constituents of the IBM ORB component.

What the ORB component contains

The ORB component contains the following:

- Java ORB from IBM and rmi-iiop runtime (com.ibm.rmi.*, com.ibm.CORBA.*)
- rmi-iiop API (javax.rmi.CORBA.*,org.omg.CORBA.*)
- IDL to Java implementation (org.omg.* and IBM versions com.ibm.org.omg.*)
- Transient name server (com.ibm.CosNaming.*, org.omg.CosNaming.*) - tnameserv
- -iiop and -idl generators (com.ibm.tools.rmi.rmic.*) for the rmic compiler - rmic
- idlj compiler (com.ibm.idl.*)

What the ORB component does not contain

The ORB component does ***not*** contain:

- RMI-JRMP (also known as Standard RMI)

- JNDI and its plug-ins

Therefore, if the problem is in java.rmi.* or sun.rmi.* , it is ***not*** an ORB problem. Similarly, if the problem is in com.sun.jndi.*, it is ***not*** an ORB problem.

Platform dependent problems

If possible, run the test case on more than one platform. All the ORB code is shared. You can nearly always reproduce genuine ORB problems on any platform. If you have a platform-specific problem, it is likely to be in some other component.

JIT problem

JIT bugs are very difficult to find. They might show themselves as ORB problems. When you are debugging or testing an ORB application, it is always safer to switch off the JIT by setting the option **-Xint**.

Fragmentation

Disable fragmentation when you are debugging the ORB. Although fragmentation does not add complications to the ORB's functioning, a fragmentation bug can be difficult to detect because it will most likely show as a general marshalling problem. The way to disable fragmentation is to set the ORB property **com.ibm.CORBA.FragmentSize=0**. You must do this on the client side and on the server side.

ORB versions

The ORB component carries a few version properties that you can display by invoking the main method of the following classes:

1. com.ibm.CORBA.iiop.Version (ORB runtime version)
2. com.ibm.tools.rmic.iiop.Version (for tools; for example, idlj and rmic)
3. rmic -iiop -version (run the command line for rmic)

Note: Items 2 and 3 are alternative methods for reaching the same class.

Limitation with bidirectional GIOP

Bidirectional GIOP is not supported.

Debug properties

Properties to use to enable ORB traces.

Attention: Do not turn on tracing for normal operation, because it might cause performance degradation. Even if you have switched off tracing, FFDC (First Failure Data Capture) is still working, so that only serious errors are reported. If a debug output file is generated, examine it to check on the problem. For example, the server might have stopped without performing an ORB.shutdown().

You can use the following properties to enable the ORB traces:

- **com.ibm.CORBA.Debug:** This property turns on trace, message, or both. If you set this property to **trace**, only traces are enabled; if set to **message**, only messages are enabled. When set to **true**, both types are enabled; when set to **false**, both types are disabled. The default is **false**.

- **com.ibm.CORBA.Debug.Output:** This property redirects traces to a file, which is known as a trace log. When this property is not specified, or it is set to an empty string, the file name defaults to the format orbtrc.DDMMYYYY.HHmm.SS.txt, where D=Day; M=Month; Y=Year; H=Hour (24 hour format); m=Minutes; S=Seconds. Note that if the application (or Applet) does not have the privilege that it requires to write to a file, the trace entries go to stderr.
- **com.ibm.CORBA.CommTrace:** This property turns on wire tracing. Every incoming and outgoing GIOP message will be output to the trace log. You can set this property independently from Debug; this is useful if you want to look only at the flow of information, and you are not too worried about debugging the internals. The only two values that this property can have are **true** and **false**. The default is **false**.

Here is an example of common usage example:

```
java -Dcom.ibm.CORBA.Debug=true -Dcom.ibm.CORBA.Debug.Output=trace.log -Dcom.ibm.CORBA.CommTrace=true <classname>
```

For rmic -iiop or rmic -idl, the following diagnostic tools are available:

- **-J-Djavac.dump.stack=1:** This tool ensures that all exceptions are caught.
- **-Xtrace:** This tool traces the progress of the parse step.

If you are working with an IBM SDK, you can obtain CommTrace for the transient name server (tnameserv) by using the standard environment variable **IBM_JAVA_OPTIONS**. In a separate command session to the server or client SDKs, you can use:

```
export IBM_JAVA_OPTIONS=-Dcom.ibm.CORBA.CommTrace=true -Dcom.ibm.CORBA.Debug=true
```

The setting of this environment variable affects each Java process that is started, so use this variable carefully. Alternatively, you can use the **-J** option to pass the properties through the tnameserv wrapper, as follows:

```
tnameserv -J-Dcom.ibm.CORBA.Debug=true
```

ORB exceptions

User and system exceptions for CORBA.

If your problem is related to the ORB, it is likely that your log file or terminal is full of exceptions that include the words “CORBA” and “rmi” many times. All unusual behavior that occurs in a good application is highlighted by an exception. This principle is also true for the ORB with its CORBA exceptions. Similar to Java, CORBA divides its exceptions into user exceptions and system exceptions.

User exceptions

User exceptions are IDL defined and inherit from org.omg.CORBA.UserException. These exceptions are mapped to checked exceptions in Java; that is, if a remote method raises one of them, the application that invoked that method must catch the exception. User exceptions are usually not fatal exceptions and should always be handled by the application. Therefore, if you get one of these user exceptions, you know where the problem is, because the application developer had to make allowance for such an exception to occur. In most of these cases, the ORB is not the source of the problem.

System exceptions

System exceptions are thrown transparently to the application and represent an unusual condition in which the ORB cannot recover gracefully, such as when a connection is dropped. The CORBA 2.6 specification defines 31 system exceptions and their mapping to Java. They all belong to the org.omg.CORBA package. The CORBA specification defines the meaning of these exceptions and describes the conditions in which they are thrown.

The most common system exceptions are:

- **BAD_OPERATION:** This exception is thrown when an object reference denotes an existing object, but the object does not support the operation that was invoked.
- **BAD_PARAM:** This exception is thrown when a parameter that is passed to a call is out of range or otherwise considered illegal. An ORB might raise this exception if null values or null pointers are passed to an operation.
- **COMM_FAILURE:** This exception is raised if communication is lost while an operation is in progress, after the request was sent by the client, but before the reply from the server has been returned to the client.
- **DATA_CONVERSION:** This exception is raised if an ORB cannot convert the marshaled representation of data into its native representation, or cannot convert the native representation of data into its marshaled representation. For example, this exception can be raised if wide character codeset conversion fails, or if an ORB cannot convert floating point values between different representations.
- **MARSHAL:** This exception indicates that the request or reply from the network is structurally not valid. This error typically indicates a bug in either the client-side or server-side runtime. For example, if a reply from the server indicates that the message contains 1000 bytes, but the actual message is shorter or longer than 1000 bytes, the ORB raises this exception.
- **NO_IMPLEMENT:** This exception indicates that although the operation that was invoked exists (it has an IDL definition), no implementation exists for that operation.
- **UNKNOWN:** This exception is raised if an implementation throws a non-CORBA exception, such as an exception that is specific to the implementation's programming language. It is also raised if the server returns a system exception that is unknown to the client. (This can happen if the server uses a later version of CORBA than the version that the client is using, and new system exceptions have been added to the later version.)

Completion status and minor codes

Each system exception has two pieces of data that are associated with it:

- A completion status, which is an enumerated type that has three values: COMPLETED_YES, COMPLETED_NO and COMPLETED_MAYBE. These values indicate either that the operation was executed in full, that the operation was not executed, or that this cannot be determined.
- A long integer, called minor code, that can be set to some ORB vendor specific value. CORBA also specifies the value of many minor codes.

Usually the completion status is not very useful. However, the minor code can be essential when the stack trace is missing. In many cases, the minor code identifies the exact location of the ORB code where the exception is thrown (see the section

below) and can be used by the vendor's service team to localize the problem quickly. However, for standard CORBA minor codes, this is not always possible. For example:

```
org.omg.CORBA.OBJECT_NOT_EXIST: SERVANT_NOT_FOUND   minor code: 4942FC11  completed: No
```

Minor codes are usually expressed in hexadecimal notation (except for SUN's minor codes, which are in decimal notation) that represents four bytes. The OMG organization has assigned to each vendor a range of 4096 minor codes. The IBM vendor-specific minor code range is 0x4942F000 through 0x4942FFFF. Corba Messages gives diagnostic information for the most-common minor codes.

System exceptions might also contain a string that describes the exception and other useful information. You will see this string when you interpret the stack trace.

The ORB tends to map all Java exceptions to CORBA exceptions. A runtime exception is mapped to a CORBA system exception, while a checked exception is mapped to a CORBA user exception.

More exceptions other than the CORBA exceptions could be generated by the ORB component in a code bug. All the Java unchecked exceptions and errors and others that are related to the ORB tools rmic and idlj must be considered. In this case, the only way to determine whether the problem is in the ORB, is to look at the generated stack trace and see whether the objects involved belong to ORB packages.

Java security permissions for the ORB

When running with a Java SecurityManager, invocation of some methods in the CORBA API classes might cause permission checks to be made that could result in a SecurityException. Here is a selection of affected methods:

Table 16. Methods affected when running with Java 2 SecurityManager

Class/Interface	Method	Required permission
org.omg.CORBA.ORB	init	java.net.SocketPermission resolve
org.omg.CORBA.ORB	connect	java.net.SocketPermission listen
org.omg.CORBA.ORB	resolve_initial_references	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_is_a	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_non_existent	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	OutputStream _request (String, boolean)	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_get_interface_def	java.net.SocketPermission connect
org.omg.CORBA. Request	invoke	java.net.SocketPermission connect
org.omg.CORBA. Request	send_deferred	java.net.SocketPermission connect

Table 16. Methods affected when running with Java 2 SecurityManager (continued)

Class/Interface	Method	Required permission
org.omg.CORBA. Request	send_oneway	java.net.SocketPermission connect
javax.rmi. PortableRemoteObject	narrow	java.net.SocketPermission connect

If your program uses any of these methods, ensure that it is granted the necessary permissions.

Interpreting the stack trace

Whether the ORB is part of a middleware application or you are using a Java standalone application (or even an applet), you must retrieve the stack trace that is generated at the moment of failure. It could be in a log file, or in your terminal or browser window, and it could consist of several chunks of stack traces.

Figure 13 shows a stack trace that was generated by a server ORB running in the WebSphere Application Server:

```
org.omg.CORBA.MARSHAL: com.ibm.ws.pmi.server.DataDescriptor; IllegalAccessException  minor code: 4942F23E  completed: No
         at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:199)
         at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
         at com.ibm.rmi.io.ValueHandlerImpl.read_Array(ValueHandlerImpl.java:625)
         at com.ibm.rmi.io.ValueHandlerImpl.readValueInternal(ValueHandlerImpl.java:273)
         at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:189)
         at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
         at com.ibm.ejs.sm.beans._EJSRemoteStatelessPmiService_Tie._invoke(_EJSRemoteStatelessPmiService_Tie.java:613)
         at com.ibm.CORBA.iiop.ExtendedServerDelegate.dispatch(ExtendedServerDelegate.java:515)
         at com.ibm.CORBA.iiop.ORB.process(ORB.java:2377)
         at com.ibm.CORBA.iiop.OrbWorker.run(OrbWorker.java:186)
         at com.ibm.ejs.oa.pool.ThreadPool$PooledWorker.run(ThreadPool.java:104)
         at com.ibm.ws.util.CachedThread.run(ThreadPool.java:137)
```

Figure 13. Example of a stack trace

In the example, the ORB mapped a Java exception to a CORBA exception. This exception is sent back to the client later as part of a reply message. The client ORB reads this exception from the reply. It maps it to a Java exception (java.rmi.RemoteException according to the CORBA specification) and throws this new exception back to the client application.

Description string

The example stack trace shows that the application has caught a CORBA org.omg.CORBA.MARSHAL system exception. After the MARSHAL exception, some extra information is provided in the form of a string. This string should specify minor code, completion status, and other information that is related to the problem. Because CORBA system exceptions are alarm bells for an unusual condition, they also hide inside what the real exception was.

Usually, the type of the exception is written in the message string of the CORBA exception. The trace shows that the application was reading a value (read_value()) when an IllegalAccessException occurred that was associated to class com.ibm.ws.pmi.server.DataDescriptor. This is a hint of the real problem and should be investigated first.

Interpreting ORB traces

The ORB trace file contains messages, trace points, and wire tracing. This section describes the various types of trace.

Message trace

An example of a message trace.

Here is a simple example of a message:

```
19:12:36.306 com.ibm.rmi.util.Version logVersions:110 P=754534:O=0:CT
ORBRas[default] IBM Java ORB build orbdev-20050927
```

This message records the time, the package, and the method name that was invoked. In this case, logVersions() prints out to the log file, the version of the running ORB.

After the first colon in the example message, the line number in the source code where that method invocation is done is written (88 in this case). Next follows the letter P that is associated with the process number that was running at that moment. This number is related (by a hash) to the time at which the ORB class was loaded in that process. It unlikely that two different processes load their ORBs at the same time.

The following O=0 (alphabetic O = numeric 0) indicates that the current instance of the ORB is the first one (number 0). CT® specifies that this is the main (control) thread. Other values are: LT for listener thread, RT for reader thread, and WT for worker thread.

The ORBRas field shows which RAS implementation the ORB is running. It is possible that when the ORB runs inside another application (such as a WebSphere application), the ORB RAS default code is replaced by an external implementation.

The remaining information is specific to the method that has been logged while executing. In this case, the method is a utility method that logs the version of the ORB.

This example of a possible message shows the logging of entry or exit point of methods, such as:

```
14:54:14.848 com.ibm.rmi.iiop.Connection <init>:504 LT=0:P=650241:O=0:port=1360 ORBRas[default] Entry
.....
14:54:14.857 com.ibm.rmi.iiop.Connection <init>:539 LT=0:P=650241:O=0:port=1360 ORBRas[default] Exit
```

In this case, the constructor (that is, <init>) of the class Connection is invoked. The tracing records when it started and when it finished. For operations that include the java.net package, the ORBRas logger prints also the number of the local port that was involved.

Comm traces

An example of comm (wire) tracing.

Here is an example of comm (wire) tracing:

```
// Summary of the message containing name-value pairs for the principal fields
OUT GOING:
Request Message // It is an out going request, therefore we are dealing with a client
Date:           31 January 2003 16:17:34 GMT
Thread Info:    P=852270:O=0:CT
Local Port:     4899 (0x1323)
Local IP:       9.20.178.136
Remote Port:    4893 (0x131D)
Remote IP:      9.20.178.136
GIOP Version:   1.2
Byte order:     big endian
```

```
Fragment to follow: No     // This is the last fragment of the request
Message size:   276 (0x114)
--

Request ID:         5  // Request Ids are in ascending sequence
Response Flag:      WITH_TARGET // it means we are expecting a reply to this request
Target Address:     0
Object Key:         length = 26 (0x1A)  // the object key is created by the server when exporting
                                        // the servant and retrieved in the IOR using a naming service
                    4C4D4249 00000010 14F94CA4 00100000
                    00080000 00000000 0000
Operation:          message // That is the name of the method that the client invokes on the servant
Service Context:    length = 3 (0x3)   // There are three service contexts
Context ID:         1229081874 (0x49424D12) // Partner version service context. IBM only
Context data:       length = 8 (0x8)
                    00000000 14000005


Context ID:         1 (0x1)  // Codeset CORBA service context
Context data:       length = 12 (0xC)
                    00000000 00010001 00010100

Context ID:         6 (0x6)  // Codebase CORBA service context
Context data:       length = 168 (0xA8)
                    00000000 00000028 49444C3A 6F6D672E
                    6F72672F 53656E64 696E6743 6F6E7465
                    78742F43 6F646542 6173653A 312E3000
                    00000001 00000000 0000006C 00010200
                    0000000D 392E3230 2E313738 2E313336
                    00001324 0000001A 4C4D4249 00000010
                    15074A96 00100000 00080000 00000000
                    00000000 00000002 00000001 00000018
                    00000000 00010001 00000001 00010020
                    00010100 00000000 49424D0A 00000008
                    00000000 14000005
Data Offset:        11c
// raw data that goes in the wire in numbered rows of 16 bytes and the corresponding ASCII
decoding
0000: 47494F50 01020000 00000114 00000005    GIOP............
0010: 03000000 00000000 0000001A 4C4D4249    ............LMBI
0020: 00000010 14F94CA4 00100000 00080000    ......L.........
0030: 00000000 00000000 00000008 6D657373    ............mess
0040: 61676500 00000003 49424D12 00000008    age.....IBM.....
0050: 00000000 14000005 00000001 0000000C    ................
0060: 00000000 00010001 00010100 00000006    ................
0070: 000000A8 00000000 00000028 49444C3A    ...........(IDL:
0080: 6F6D672E 6F72672F 53656E64 696E6743    omg.org/SendingC
0090: 6F6E7465 78742F43 6F646542 6173653A    ontext/CodeBase:
00A0: 312E3000 00000001 00000000 0000006C    1.0............l
00B0: 00010200 0000000D 392E3230 2E313738    ........9.20.178
00C0: 2E313336 00001324 0000001A 4C4D4249    .136...$....LMBI
00D0: 00000010 15074A96 00100000 00080000    ......J.........
00E0: 00000000 00000000 00000002 00000001    ................
00F0: 00000018 00000000 00010001 00000001    ................
0100: 00010020 00010100 00000000 49424D0A    ... ........IBM.
0110: 00000008 00000000 14000005 00000000    ................
```

Note: The italic comments that start with a double slash have been added for clarity; they are not part of the traces.

In this example trace, you can see a summary of the principal fields that are contained in the message, followed by the message itself as it goes in the wire. In the summary are several field name-value pairs. Each number is in hexadecimal notation.

For details of the structure of a GIOP message, see the CORBA specification, chapters 13 and 15.

Client or server

From the first line of the summary of the message, you can identify whether the host to which this trace belongs is acting as a server or as a client. OUT GOING means that the message has been generated in the machine where the trace was taken and is sent to the wire.

In a distributed-object application, a server is defined as the provider of the implementation of the remote object to which the client connects. In this work, however, the convention is that a client sends a request while the server sends back a reply. In this way, the same ORB can be client and server in different moments of the rmi-iiop session.

The trace shows that the message is an outgoing request. Therefore, this trace is a client trace, or at least part of the trace where the application acts as a client.

Time information and host names are reported in the header of the message.

The Request ID and the Operation ("message" in this case) fields can be very helpful when multiple threads and clients destroy the logical sequence of the traces.

The GIOP version field can be checked if different ORBs are deployed. If two different ORBs support different versions of GIOP, the ORB that is using the more recent version of GIOP should fall back to a common level. By checking that field, however, you can easily check whether the two ORBs speak the same language.

Service contexts

The header also records three service contexts, each consisting of a context ID and context data.

A service context is extra information that is attached to the message for purposes that can be vendor-specific (such as the IBM Partner version that is described in the IOR in *Diagnostics Guide*).

Usually, a security implementation makes extensive use of these service contexts. Information about an access list, an authorization, encrypted IDs, and passwords could travel with the request inside a service context.

Some CORBA-defined service contexts are available. One of these is the Codeset.

In the example, the codeset context has `ID 1` and data `00000000 00010001 00010100`. Bytes 5 through 8 specify that characters that are used in the message are encoded in ASCII (00010001 is the code for ASCII). Bytes 9 through 12 instead are related to wide characters.

The default codeset is UTF8 as defined in the CORBA specification, although almost all Windows and UNIX® platforms communicate normally through ASCII. i5/OS® and Mainframes such as zSeries® systems are based on the IBM EBCDIC encoding.

The other CORBA service context, which is present in the example, is the Codebase service context. It stores information about how to call back to the client to access

resources in the client such as stubs, and class implementations of parameter objects that are serialized with the request.

Common problems

This section describes some of the problems that you might find.

ORB application hangs

One of the worst conditions is when the client, or server, or both, hang. If this happens, the most likely condition (and most difficult to solve) is a deadlock of threads. In this condition, it is important to know whether the machine that on which you are running has more than one CPU, and whether your CPU is using Simultaneous Multithreading (SMT).

A simple test that you can do is to keep only one CPU running, disable SMT, and see whether the problem disappears. If it does, you know that you must have a synchronization problem in the application.

Also, you must understand what the application is doing while it hangs. Is it waiting (low CPU usage), or it is looping forever (almost 100% CPU usage)? Most of the cases are a waiting problem.

You can, however, still identify two cases:

- Typical deadlock
- Standby condition while the application waits for a resource to arrive

An example of a standby condition is where the client sends a request to the server and stops while waiting for the reply. The default behavior of the ORB is to wait indefinitely.

You can set a couple of properties to avoid this condition:

- com.ibm.CORBA.LocateRequestTimeout
- com.ibm.CORBA.RequestTimeout

When the property com.ibm.CORBA.enableLocateRequest is set to true (the default is false), the ORB first sends a short message to the server to find the object that it needs to access. This first contact is the Locate Request. You must now set the LocateRequestTimeout to a value other than 0 (which is equivalent to infinity). A good value could be something around 5000 milliseconds.

Also, set the RequestTimeout to a value other than 0. Because a reply to a request is often large, allow more time; for example, 10000 milliseconds. These values are suggestions and might be too low for slow connections. When a request times out, the client receives an explanatory CORBA exception.

When an application hangs, consider also another property that is called com.ibm.CORBA.FragmentTimeout. This property was introduced in IBM ORB 1.3.1, when the concept of fragmentation was implemented to increase performance. You can now split long messages into small chunks or fragments and send one after the other across the net. The ORB waits for 30 seconds (default value) for the next fragment before it throws an exception. If you set this property, you disable this time-out, and problems of waiting threads might occur.

If the problem appears to be a deadlock or hang, capture the Javadump information. Do this once, then wait for a minute or so, and do it again. A

comparison of the two snapshots shows whether any threads have changed state. For information about how to do this operation, see "Triggering a Javadump" on page 122.

In general, stop the application, enable the orb traces (see previous section) and restart the application. When the hang is reproduced, the partial traces that can be retrieved can be used by the IBM ORB service team to help understand where the problem is.

Running the client without the server running before the client is invoked

An example of the error messages that are generated from this process.

This operation outputs:

```
(org.omg.CORBA.COMM_FAILURE)
Hello Client exception:
        org.omg.CORBA.COMM_FAILURE:minor code:1 completed:No
             at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:145)
             at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:77)
             at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:98)
             at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:75)
             at com.ibm.rmi.corba.ClientDelegate.createRequest(ClientDelegate.java:440)
             at com.ibm.rmi.corba.ClientDelegate.is_a(ClientDelegate.java:571)
             at org.omg.CORBA.portable.ObjectImpl._is_a(ObjectImpl.java:74)
             at org.omg.CosNaming.NamingContextHelper.narrow(NamingContextHelper.java:58)
             com.sun.jndi.cosnaming.CNCtx.callResolve(CNCtx.java:327)
```

Client and server are running, but not naming service

An example of the error messages that are generated from this process.

The output is:

```
Hello Client exception:Cannot connect to ORB
Javax.naming.CommunicationException:
        Cannot connect to ORB.Root exception is org.omg.CORBA.COMM_FAILURE  minor code:1 completed:No
               at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:145)
               at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:77)
               at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:98)
               at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:75)
               at com.ibm.rmi.corba.ClientDelegate.createRequest(ClientDelegate.java:440)
               at com.ibm.rmi.corba.InitialNamingClient.resolve(InitialNamingClient.java:197)
               at com.ibm.rmi.corba.InitialNamingClient.cachedInitialReferences(InitialNamingClient.j
               at com.ibm.rmi.corba.InitialNamingClient.resolve_initial_references(InitialNamingClien
               at com.ibm.rmi.corba.ORB.resolve_initial_references(ORB.java:1269)
               .........
```

You must start the Java IDL name server before an application or applet starts that uses its naming service. Installation of the Java IDL product creates a script or executable file that starts the Java IDL name server.

Start the name server so that it runs in the background. If you do not specify otherwise, the name server listens on port 2809 for the bootstrap protocol that is used to implement the ORB resolve_initial_references() and list_initial_references() methods.

Specify a different port, for example, 1050, as follows:

```
tnameserv -ORBInitialPort 1050
```

Clients of the name server must be made aware of the new port number. Do this by setting the org.omg.CORBA.ORBInitialPort property to the new port number when you create the ORB object.

Running the client with MACHINE2 (client) unplugged from the network

An example of the error messages that are generated when the client has been unplugged form the network.

Your output is:

```
(org.omg.CORBA.TRANSIENT CONNECT_FAILURE)

Hello Client exception:Problem contacting address:corbaloc:iiop:machine2:2809/NameService
javax.naming.CommunicationException:Problem contacting address:corbaloc:iiop:machine2:2809/N
      is org.omg.CORBA.TRANSIENT:CONNECT_FAILURE (1)minor code:4942F301 completed:No
              at com.ibm.CORBA.transport.TransportConnectionBase.connect(TransportConnectionBase.jav
              at com.ibm.rmi.transport.TCPTransport.getConnection(TCPTransport.java:178)
              at com.ibm.rmi.iiop.TransportManager.get(TransportManager.java:79)
              at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:131)
              at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:98)
              at com.ibm.CORBA.iiop.ClientDelegate._createRequest(ClientDelegate.java:2096)
              at com.ibm.CORBA.iiop.ClientDelegate.createRequest(ClientDelegate.java:1264)
              at com.ibm.CORBA.iiop.ClientDelegate.createRequest(ClientDelegate.java:1177)
              at com.ibm.rmi.corba.InitialNamingClient.resolve(InitialNamingClient.java:252)
              at com.ibm.rmi.corba.InitialNamingClient.cachedInitialReferences(InitialNamingClient.j
              at com.ibm.rmi.corba.InitialNamingClient.resolve_initial_references(InitialNamingClien
              at com.ibm.rmi.corba.InitialReferenceClient.resolve_initial_references(InitialReferenc
              at com.ibm.rmi.corba.ORB.resolve_initial_references(ORB.java:3211)
              at com.ibm.rmi.iiop.ORB.resolve_initial_references(ORB.java:523)
              at com.ibm.CORBA.iiop.ORB.resolve_initial_references(ORB.java:2898)
              ..........
```

IBM ORB service: collecting data

This section describes how to collect data about ORB problems.

If after all these verifications, the problem is still present, collect at all nodes of the problem the following:

- Operating system name and version.
- Output of `java -Xrealtime -version`.
- Output of `java com.ibm.CORBA.iiop.Version`.
- Output of `rmic -iiop -version`, if rmic is involved.
- ASV build number (WebSphere Application Server only).
- If you think that the problem is a regression, include the version information for the most recent known working build and for the failing build.
- If this is a runtime problem, collect debug and communication traces of the failure from each node in the system (as explained earlier in this section).
- If the problem is in `rmic -iiop` or `rmic -idl`, set the options: `-J-Djavac.dump.stack=1 -Xtrace`, and capture the output.
- Normally this step is not necessary. If it looks like the problem is in the buffer fragmentation code, IBM service will return the defect asking for an additional set of traces, which you can produce by executing with `-Dcom.ibm.CORBA.FragmentSize=0`.

A testcase is not essential, initially. However, a working testcase that demonstrates the problem by using only the Java SDK classes will speed up the resolution time for the problem.

Preliminary tests

The ORB is affected by problems with the underlying network, hardware, and JVM.

When a problem occurs, the ORB can throw an org.omg.CORBA.* exception, some text that describes the reason, a minor code, and a completion status. Before you assume that the ORB is the cause of problem, ensure the following:

- The scenario can be reproduced in a similar configuration.
- The JIT is disabled (see "Problem determination for compilers" on page 154).
- No AOT compiled code is being used

Also:

- Disable additional CPUs.
- Disable Simultaneous Multithreading (SMT) where possible.
- Eliminate memory dependencies with the client or server. The lack of physical memory can be the cause of slow performance, apparent hangs, or crashes. To remove these problems, ensure that you have a reasonable headroom of memory.
- Check physical network problems (firewalls, comm links, routers, DNS name servers, and so on). These are the major causes of CORBA COMM_FAILURE exceptions. As a test, ping your own machine name.
- If the application is using a database such as DB2®, switch to the most reliable driver. For example, to isolate DB2 AppDriver, switch to Net Driver, which is slower and uses sockets, but is more reliable.

NLS problem determination

The JVM contains built-in support for different locales. This section provides an overview of locales, with the main focus on fonts and font management.

Overview of fonts

When you want to display text, either in SDK components (AWT or Swing), on the console or in any application, characters have to be mapped to **glyphs**.

A glyph is an artistic representation of the character, in some typographical style, and is stored in the form of outlines or bitmaps. Glyphs might not correspond one-for-one with characters. For instance, an entire character sequence can be represented as a single glyph. Also a single character may be represented by more than one glyph (for example, in Indic scridts).

A font is a set of glyphs, where each glyph is encoded in a particular encoding format, so that the character to glyph mapping can be done using the encoded value. Almost all of the available Java fonts are encoded in Unicode and provide universal mappings for all applications.

The most commonly available font types are TrueType and OpenType fonts.

Font specification properties

Specify fonts according to the following characteristics:

Font family
A font family is a group of several individual fonts that are related in appearance. For example: Times, Arial, and Helvetica.

Font style
Font style specifies that the font be displayed in various faces. For example: Normal, Italic, and Oblique

Font variant
This property determines whether the font should be displayed in normal caps or in small caps. A particular font might contain only normal caps, only small caps, or both types of glyph.

Font weight
This refers to the boldness or the lightness of the glyph to be used.

Font size
This property is used to modify the size of the displayed text.

Fonts installed in the system

To see the fonts that are either installed in the Linux system or available for an application to use, type the command: `xset -q ""`. If your PATH also points to the SDK (as it should be), `xset -q` output also shows the fonts that are bundled with the Developer Kit.

Use `xset +fp` and `xset -fp` to add and remove the font path respectively.

Font utilities

A list of font utilities that are supported.

xlsfonts
Use **xlsfonts** to check whether a particular font is installed on the system. For example: `xlsfonts | grep ksc` will list all the Korean fonts in the system.

iconv Use to convert the character encoding from one encoding to other. Converted text is written to standard output. For example: `iconv -f oldset -t newset [file ...]`

Odtions are:

-f oldset
Specifies the source codeset (encoding).

-t newset
Specifies the destination codeset (encoding).

file The file that contain the characters to be converted; if no file is specified, standard input is used.

Common problems and possible causes

A list of common NLS problems with potential solutions.

Why do I see a square box or ??? (question marks) in the SDK components?
This effect is caused mainly because Java is not able to find the correct font file to display the character. If a Korean character should be displayed, the system should be using the Korean locale, so that Java can take the correct font file. If you are seeing boxes or queries, check the following:

For AWT components:

1. Check your locale with `locale`.
2. To change the locale, export `LANG=zh_TW` (for example)
3. If this still does not work, try to log in with the required language.

For Swing components:

1. Check your locale with `locale`
2. To change the locale, export `LANG=zh_TW` (for example)
3. If you know which font you have used in your application, such as serif, try to get the corresponding physical font by looking in the fontpath. If the font file is missing, try adding it there.

Chapter 11. Using diagnostic tools

This part of the book describes how to use the diagnostic tools that are available. The sections are:

- "Using dump and trace agents"
- "Using Javadump" on page 121
- "Using core (system) dumps" on page 134
- "Using method trace" on page 137
- "Problem determination for compilers" on page 154
- "Garbage Collector diagnostics" on page 158

Note:

1. JVMMI is not supported on the Version 5 platforms.
2. JVMPI is now a deprecated interface, replaced by JVMTI.

Using dump and trace agents

Dump agents are set up during JVM initialization. They enable you to use events occurring within the JVM, such as Garbage Collection, thread start, or JVM termination, to initiate one of five types of dump (console, system, java, heap, and snap) or to launch an external tool.

Default dump agents are set up at JVM initialization. They are sufficient for most cases, but the use of the **-Xdump** option on the command line allows more detailed configuration of dump agents. The total set of options and suboptions available under **-Xdump** is very flexible and there are many examples presented in this section to show this flexibility.

The **-Xdump** option allows you to add and remove dump agents for various JVM events, update default dump settings (such as the dump name), and limit the number of dumps that are produced.

This section describes:

- "Help options"
- "Dump types and triggering" on page 113
- "Types of dump agents - examples" on page 115
- "Default dump agents" on page 117
- "Default settings for dumps" on page 118
- "Limiting dumps using filters and range keywords" on page 119
- "Removing dump agents" on page 119
- "Controlling dump ordering" on page 119
- "Controlling dump file names" on page 120

Help options

You can obtain help on the various usage aspects of **-Xdump** by using **java -Xrealtime -Xdump:help**.

Table 17. Usage from java -Xdump:help

Command	Result
-Xdump:help	Print general dump help
-Xdump:none	Ignore all previous and default dump options
-Xdump:events	List available trigger events
-Xdump:request	List additional VM requests
-Xdump:tokens	List recognized label tokens
-Xdump:dynamic	Enable support for pluggable agents
-Xdump:what	Show registered agents on startup
-Xdump:<type>:help	Print detailed dump help
-Xdump:<type>:none	Ignore previous dump options of this type
-Xdump:<type>:defaults	Print and update default settings for this type
-Xdump:<type>	Request this type of dump (using defaults)

Table 18. Types of dump

Valid types of dump	Description
-Xdump:console	Basic thread dump to stderr
-Xdump:system	Capture raw process image. See “Using core (system) dumps” on page 134.
-Xdump:tool	Run command line program
-Xdump:java	Write application summary. See “Using Javadump” on page 121.
-Xdump:snap	Take a snap of the trace buffers

Example

```
java -Xdump:heap:none -Xdump:heap:events=fullgc class [args...]
```

turns off default Heapdumps and then requests a Heapdump on every full GC.

As can be seen from Table 17, further help is available for the assorted suboptions under **-Xdump**. In particular, **java -Xdump:events** shows the available keywords used to specify the events that can be used.

You must filter class events (such as load, throw, and uncaught) by class name. For guidance, see “Limiting dumps using filters and range keywords” on page 119.

Note: The unload and expand events currently do not occur in WebSphere Real Time. Classes are in immortal memory and cannot be unloaded.

Table 19. Keywords

Supported event keywords	Event hook
gpf	ON_GP_FAULT
user	ON_USER_SIGNAL
abort	ON_ABORT_SIGNAL
vmstart	ON_VM_STARTUP
vmstop	ON_VM_SHUTDOWN

Table 19. Keywords (continued)

Supported event keywords	Event hook
load	ON_CLASS_LOAD
unload (Not applicable for WebSphere Real Time)	ON_CLASS_UNLOAD
throw	ON_EXCEPTION_THROW
catch	ON_EXCEPTION_CATCH
brkpoint	ON_BREAKPOINT
framepop	ON_DEBUG_FRAME_POP
thrstart	ON_THREAD_START
blocked	ON_THREAD_BLOCKED
thrstop	ON_THREAD_END
expand (Not applicable for WebSphere Real Time)	ON_HEAP_EXPAND
fullgc	ON_GLOBAL_GC
uncaught	ON_EXCEPTION_DESCRIBE
slow	ON_SLOW_EXCLUSIVE_ENTER
any	*

Dump types and triggering

The main purpose of the **-Xdump** option on the command line is to link *events* to a *dump type* (**-Xdump:tool** is a little misleading, because it is a command, not a dump).

-Xdump:heap:events=vmstop is an instruction to JVM initialization to create a dump agent that produces a Heapdump whenever the vmstop event happens. The JVM is constructed to generate at the appropriate time the events listed in "Using core (system) dumps" on page 134.

You can have multiple **-Xdump** options on the command line and also multiple dump types driven by one or multiple events. Thus, **-Xdump:heap+java:events=vmstart+vmstop** would create a dump agent that would drive both heap and Java dump production when either a vmstart or vmstop event was encountered. Multiple commands can be used in a **-Xdump** option by separating each command with a comma. For example: **-Xdump:heap:event=vmstart,opts=CLASSIC** creates a heapdump in the classic format when the JVM starts.

Note: Multiple **-Xdump** options on the command line can be used to create multiple agents at JVM initialization; these agents are chained together and all evaluated whenever an event occurs. The dump agent processing ensures that multiple **-Xdump** options are optimized. You can use the **-Xdump:what** option to clarify this optimization.

The keyword **events** is used as the prime trigger mechanism. However, there are a number of additional keywords that you can use to further control the dump

produced, for example, **request** and **tokens** or limit its production to a smaller range of circumstances; use **-Xdump:<type>:help** to find these.

Obtaining JVM trace information

It is valuable to get low-level information about what the virtual machine is doing. This information is used to determine the flow of execution of the application or to gather information about where an application is spending time.

Note: Running with the **-Xtrace** option can significantly affect the performance of your application as it gathers trace information, especially if you perform maximal tracing.

Initially you run the application with the **-Xtrace** option to generate a binary file, then you post-process the file to generate a human-readable version.

To get a maximum trace of every available trace point, use the following **-Xtrace** option:

1. **-Xtrace:maximal=all,output=***filename.trace*. The **-Xtrace** option and output from the trace formatter provide JVM implementation details which are subject to change between releases. All information provided by **-Xtrace** should be used for general program understanding and not as input to any programmatic interface.
2. Run the post-processing tool: `java com.ibm.jvm.format.TraceFormat filename.trace filename.out` The file *filename.out* has trace records for each event traced. To generate a smaller trace file, you can restrict the output to a set of components. The syntax is:

   ```
   -Xtrace:maximal={component,component,...,component}
   ```

   where *component* specifies the JVM component you want to trace.

   Components of most interest are:
   j9vm: Virtual Machine
   j9mm: Memory Manager (Garbage Collection is part of this)
   j9jit: Just In Time Compiler
   j9jcl: Core class libraries

For example, if you wanted to trace just the j9vm and j9mm components, you can specify:

```
-Xtrace:maximal={j9vm,j9mm},output=my.trc
```

A useful feature of tracing the j9mm component is that you can trace object allocations and, in particular, trace object allocation failures.

Object allocation entries can be:

```
14:12:09.046810000*08859300 j9mm.80 Event J9AllocateObject() returning NULL!
          32 bytes requested for object of class 0x84fbf10 from memory space 'Immortal'
```

Where *space* can be:

- *Metronome* which is the Java Heap.
- *Immortal* which is the immortal memory.
- *Scoped* which is the scoped memory.

Formatting JVM trace

The trace engine can produce data directly to the screen as it appears (the 'print' and 'iprint' options on trace) but more commonly, trace is written to a proprietary format binary file (the 'output=<filename> option).

Snap traces are in the standard trace data format and can be formatted like any other trace output file.

To get a human-readable trace out of this, use this command line:

```
java com.ibm.jvm.format.TraceFormat filename.trace filename.out
```

The file *filename.out* has trace records for each event traced in a human readable form.

Types of dump agents - examples

There are several examples of the use of the **-Xdump** option, based around each dump type, illustrating the style of syntax and the generated function. The examples given are deliberately kept simple to reduce the size of the output.

Using **java -Xrealtime -Xdump:help**, shows that there are many dump types to consider; see "Help options" on page 111.

Console dumps

Console dumps are simple dumps, in which the status of every Java thread is written to stderr.

Some output of this type is shown below. Note the use of the **range=1..1** suboption to control the amount of output to just one thread start and stop (in this case, the start of the Signal Dispatcher thread).

```
java -Xdump:console:events=thrstart+thrstop,range=1..1

JVMDUMP006I Processing Dump Event "thrstart", detail "" - Please Wait.
-------- Console dump --------
Stack Traces of Threads:
ThreadName=Signal Dispatcher(00035B24)
Status=Running
ThreadName=main(00035A1C)
Status=Waiting
Monitor=00035128 (VM sig quit)
Count=0
Owner=(00000000)
^^^^^^^^ Console dump ^^^^^^^^
```

Note: Two threads are displayed in the dump because the main thread does not generate a thrstart event.

System dumps

System dumps involve dumping a whole frozen address space and as such are generally very large. The bigger the footprint of an application the bigger its dump.

A dump of a major server-based application might take up many gigabytes of file space and take several minutes to complete. Shown below is an example of invoking a system dump on a Linux machine. Note the use of **request=serial+exclusive** in this example, to ensure that this dump is not

interrupted by other dumps, enabling the objects within the heap to be processed under **jextract** or **jdmpview**. Note also that the file name is overridden from the default in this example.

```
java -Xdump:system:events=vmstop,request=serial+exclusive,file=my.dmp

  ::::::::: removed usage info :::::::::

JVMDUMP006I Processing Dump Event "vmstop", detail "#00000000" - Please Wait.
JVMDUMP007I JVM Requesting System Dump using '/home/test/my.dmp'
JVMDUMP010I System Dump written to /home/test/my.dmp
JVMDUMP013I Processed Dump Event "vmstop", detail "#00000000".
```

gdb <*full directory for java*> <*core file*> can be used to launch **gdb** on the generated system dump file.

Tool option

The **tool** option allows external processes to be spawned when an event occurs.

Consider the following simple example, which displays start of pid and end of pid information. Note the use of the token %pid; the list of available tokens can be printed with **-Xdump:tokens**. More realistic examples would invoke a debugging tool, and that is the default taken if you use (for example) **-Xdump:tool:events=.....**

```
java -Xdump:tool:events=vmstop,exec="echo %pid has finished"
-Xdump:tool:events=vmstart,exec="echo %pid has started"
   JVMDUMP006I Processing Dump Event "vmstart", detail "" - Please Wait.
   JVMDUMP007I JVM Requesting Tool Dump using 'echo 2184 has started'
   JVMDUMP011I Tool Dump spawned process 2160
   2184 has started
   JVMDUMP013I Processed Dump Event "vmstart", detail "".

       ::::::::: removed usage info :::::::::

   JVMDUMP006I Processing Dump Event "vmstop", detail "#00000000" - Please Wait.
   JVMDUMP007I JVM Requesting Tool Dump using 'echo 2184 has finished'
   JVMDUMP011I Tool Dump spawned process 2204
   2184 has finished
   JVMDUMP013I Processed Dump Event "vmstop", detail "#00000000".
```

Javadumps

Java dumps are an internally generated and formatted analysis of the JVM, giving information that includes the Java threads present, the classes loaded, and heap statistics. Javadumps describe the state of the JVM at the point the dump was taken and are in human readable format.

An example (which also shows the use of the **filter** keyword) in which a Javadump is produced on the loading of a class is shown below.

```
java -Xdump:java:events=load,filter=*String

JVMDUMP006I Processing Dump Event "load", detail "java/lang/String" - Please Wait.
JVMDUMP007I JVM Requesting Java Dump using
 /home/test/javacore.20051012.162700.2836.txt
JVMDUMP010I Java Dump written to
 /home/test/javacore.20051012.162700.2836.txt
JVMDUMP013I Processed Dump Event "load", detail "java/lang/String".
```

Heapdumps

Heapdumps are a snapshot of the contents of one or more of the Memory Areas within the JVM. For each object in the heap a record is written to the Heapdump

file recording the location, size, and Java classname of each object, and also listing all references to other objects. Heapdumps can be viewed using, for example, MDD4J.

By default the JVM produces one heapdump containing objects from all memory areas. You can request a heapdump for each memory area by using the **-Xdump:heap:request=multiple**.

The example below shows the production of a Heapdump.

```
java -Xdump:heap:events=uncaught,filter=java/lang/OutOfMemoryError myOutOfMem

JVMDUMP006I Processing Dump Event "uncaught", detail "java/lang/OutOfMemoryError" - Please Wait.
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump.20060721.151350.13013.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump.20060721.151350.13013.phd
JVMDUMP013I Processed Dump Event "uncaught", detail "java/lang/OutOfMemoryError".
```

Snap traces

Snap traces join the range of outputs controlled by **-Xdump**.

Snap trace files are binary trace files and have the format of: `Snap%seq.%Y%m%d.%H%M%S.%pid.trc`. They contain only a small number of events; typically what was happening just before a failure.

The example below shows the production of a snap trace.

```
java -Xdump:none -Xdump:snap:events=vmstop+vmstart

JVMDUMP006I Processing Dump Event "vmstart", detail "" - Please Wait.
JVMDUMP007I JVM Requesting Snap Dump using
 '/home/test/Snap0001.20051012.161706.2804.trc'
JVMDUMP010I Snap Dump written to
 /home/test/Snap0001.20051012.161706.2804.trc
JVMDUMP013I Processed Dump Event "vmstart", detail "".

Usage: java [-options] class [args...]
           (to execute a class)
  ====  extraneous lines removed for terseness ====
    -assert   print help on assert options

JVMDUMP006I Processing Dump Event "vmstop", detail "#00000000" - Please Wait.
JVMDUMP007I JVM Requesting Snap Dump using
 '/home/test/Snap0002.20051012.161706.2804.trc'
JVMDUMP010I Snap Dump written to
 /home/test/Snap0002.20051012.161706.2804.trc
JVMDUMP013I Processed Dump Event "vmstop", detail "#00000000".
```

Snap traces are given sequential numbers (`Snap0001` then `Snap0002`). The produced snap traces are encoded and require the use of the trace formatter for further analysis.

Default dump agents

The JVM adds a set of dump agents by default during its initialization.

You can override this set of dump agents using the **JAVA_DUMP_OPTS** environment variable and further override the set by the use of **-Xdump** on the command line (See “Removing dump agents” on page 119).

The **-Xdump:what** option on the command line is very useful for determining which dump agents exist at the completion of startup and can help resolve issues

about what has overridden what. Below is sample output showing the default dump agents that are in place when there have been no overrides by using environment variables.

```
java -Xdump:what

Registered dump agents
----------------------
dumpFn=doSystemDump
events=gpf+abort
filter=
label=/home/test/core.%Y%m%d.%H%M%S.%pid.dmp
range=1..0
priority=999
request=serial
opts=
----------------------
dumpFn=doSnapDump
events=gpf+abort
filter=
label=/home/test/Snap%seq.%Y%m%d.%H%M%S.%pid.trc
range=1..0
priority=500
request=serial
opts=
----------------------
dumpFn=doSnapDump
events=uncaught
filter=java/lang/OutOfMemoryError
label=C:/home/test/Snap%seq.%Y%m%d.%H%M%S.%pid.trc
range=1..4
priority=500
request=serial
opts=----------------------
dumpFn=doHeapDump
events=uncaught
filter=java/lang/OutOfMemoryError
label=/home/test/heapdump.%Y%m%d.%H%M%S.%pid.phd
range=1..4
priority=40
request=exclusive+prepwalk
opts=PHD
----------------------
dumpFn=doJavaDump
events=gpf+user+abort
filter=
label=/home/test/javacore.%Y%m%d.%H%M%S.%pid.txt
range=1..0
priority=10
request=exclusive
opts=
----------------------
dumpFn=doJavaDump
events=uncaught
filter=java/lang/OutOfMemoryError
label=/home/test/javacore.%Y%m%d.%H%M%S.%pid.txt
range=1..4
priority=10
request=exclusive
opts=
----------------------
```

Default settings for dumps

To view the default settings for a particular dump type, use the **-Xdump:<type>:defaults** option.

You can change these defaults at runtime. For example, you can direct Java dump files into a separate directory for each process, and guarantee unique files by adding a sequence number to the file name using:

```
-Xdump:java:defaults:file=dumps/%pid/javacore-%seq.txt
```

This option does not add a javadump agent; it updates the default settings for dump agents. Further dump agents will then create dump files using this specification for filenames, unless overridden.

Limiting dumps using filters and range keywords

Some JVM events occur thousands of times during the lifetime of an application. Dump agents can use filters and ranges to avoid excessive dumps being produced.

You can filter class events (such as load, throw, and uncaught) by class name:

```
-Xdump:java:events=throw,filter=java/lang/OutOfMem* # prefix
-Xdump:java:events=throw,filter=*MemoryError # suffix
-Xdump:java:events=throw,filter=*Memory* # substring
```

You can filter the JVM shutdown event by using one or more exit codes:

```
-Xdump:java:events=vmstop,filter=#129..192#-42#255
```

You can start and stop dump agents on a particular occurrence of a JVM event by using the range suboption:

```
-Xdump:java:events=fullgc,range=100..200
```

Note that **range=1..0** against an event means ″on every occurrence″.

Users of UNIX style shells should be aware that unwanted shell expansion might occur because of the characters used in the dump agent options. To avoid unpredictable results, you should enclose this command line option in quotes. For example:

```
java "-Xdump:java:events=throw,filter=*Memory*" <Class>
```

For more information, please see the manual for your shell.

Removing dump agents

You can remove all default dump agents and any preceding dump options by using the **-Xdump:none** option.

Use this option so that you can subsequently specify a completely new dump configuration.

You can also remove dump agents of a particular type. For example,

```
-Xdump:java+heap:events=vmstop -Xdump:heap:none
```

turns off all heapdumps (including default agents) but leaves javadump enabled.

Controlling dump ordering

In some situations, one event can generate multiple dumps. For example, examination of the output from `java -Xdump:what` shows that a gpf event produces a snap trace, a java dump, and a system dump. The agents that produce each dump run sequentially and their order is determined by the priority keyword set for each agent. In the example, the system dump would run first (priority 999), the snap dump second (priority 500), and the javadump last (priority 10).

An example of setting the priority from the command line is:

```
java -Xdump:heap:events=vmstop,priority=123
```

The maximum value allowed for priority is 999 and the higher the priority the higher a dump agent will sit in the chain.

If you do not specifically set a priority then default values are taken based on the dump type. The default priority, as well as the other default values for a particular type of dump, can be displayed by using the defaults option on the **-Xdump** invocation, for example:

```
java -Xdump:heap:defaults

Default -Xdump:heap settings:

  events=gpf+user
  filter=
  file=/home/test/heapdump.%Y%m%d.%H%M%S.%pid.phd
  range=1..0
  priority=40
  request=exclusive+prepwalk
  opts=PHD
```

Controlling dump file names

Dumps are created by default in the working directory. Most often, this directory is the one from which the application was launched. If the dump cannot be created there for any reason (such as it being a read-only location), alternative locations are tried.

The pattern in the file setting for that particular dump agent determines the name of the dump. The defaults (enter: `java -Xdump:<type>:defaults`) are usually sufficient; however, you can use the file keyword on the command line to set your own file name and location.

Using Javadump

Javadump produces files that contain diagnostic information related to the JVM and a Java application captured at a point during execution. For example, the information can be about the operating system, the application environment, threads, stacks, locks, and memory.

The files produced by Javadump are called ″Javadump files″. By default, a Javadump occurs when the JVM terminates unexpectedly. A Javadump can also be triggered by sending specific signals to the JVM. Javadumps are human readable and it is often the prime tool used in diagnosing application failure.

The preferred way to control the production of Javadumps is by enabling dump agents (see “Using dump and trace agents” on page 111) using **-Xdump:java:** on application startup. You can also control Javadumps by the use of environment variables. See “Environment variables and Javadump” on page 129.

Default agents are in place that (if not overridden) create Javadumps when the JVM terminates unexpectedly or when an OutOfMemoryError condition occurs. Javadumps are also triggered by default when specific signals are received by the JVM.

Note: **Javadump** is also known as **Javacore**. This is NOT the same as a **core file** (that is an operating system feature that can be produced by any program, not just the JVM).

This section describes:

Enabling a Javadump

Javadumps are enabled by default. Javadump production can be turned off using **-Xdump:java:none**. This is not recommended as Javadumps are an essential diagnostics tool.

Use the **-Xdump:java** option to give more fine-grained control over the production of Javadumps. See “Using dump and trace agents” on page 111 for more information.

Location of the generated Javadump

When a Javadump is triggered, the JVM checks each of the following locations for existence and write-permission, and stores the Javadump in the first one available. You must have enough free disk space (possibly up to 2.5 MB) for the Javadump file to be written correctly.

1. The location specified by the **IBM_JAVACOREDIR** environment variable if set.
2. The current working directory of the JVM processes.
3. The location specified by the **TMPDIR** environment variable, if set.
4. The /tmp directory.
5. If the Javadump cannot be stored in any of the above, it is put to STDERR.

The file name is of the following form: `javacore.%Y%m%d.%H%M%S.%pid.txt` (where %pid is the process ID).

Triggering a Javadump

The exact conditions in which you get a Javadump vary depending on whether the default dump agents have been overridden.

By default, a Javadump is triggered when one of the following occurs:

- **A fatal native exception** occurs in the JVM (not a Java Exception).
- **The JVM has insufficient Java heap memory to continue operation.** Often caused by heap expansion and compaction.
- **You send a signal** to the JVM from the operating system.
- **You use the JavaDump() method** within Java code that is being executed.

A "fatal" exception is one that causes the JVM to terminate. The JVM handles this by producing a system dump followed by a snap trace file, a Javadump, and then terminating the process.

In the user-controlled cases (the latter two), the JVM pauses, performs the dump, and then continues execution.

The signal for Linux is `SIGQUIT`. Use the command `kill -3 n` to send the signal to a process with process id (PID) n. Alternatively, press **CTRL+** in the shell window that started Java.

The com.ibm.jvm.Dump class contains a static JavaDump() method that causes Java code to initiate a Javadump. In your application code, add a call to com.ibm.jvm.Dump.JavaDump(). This call is subject to the same Javadump environment variables as are described in "Enabling a Javadump" on page 121.

Interpreting a Javadump

This section gives examples of the information contained in a Javadump and how it can be useful in problem solving.

The information in a Javadump file is essentially the same, regardless of the platform on which you are running the JVM. However, certain platforms might provide more information about fatal exceptions.

The content and range of information in a Javadump is **not guaranteed across releases**. In some cases, information might be missing because of the nature of a crash.

Javadump tags

The Javadump file contains sections separated by eyecatcher title areas to aid readability of the Javadump.

The first such eyecatcher is shown below:

```
0SECTION TITLE subcomponent dump routine
NULL     ===============================
```

Different sections contain different tags, which make the file easier to parse for performing simple analysis. An example tag (`1CIJAVAVERSION`) is shown below:

```
1CIJAVAVERSION J2RE 5.0 IBM J9 2.3 Linux x86-32 build 20060727_07291_lHdRRr (JIT enabled - 20060721_1813_r8.rt)
```

Normal tags have these characteristics:

- Tags are up to 15 characters long (padded with spaces).
- The first digit is a nesting level (0,1,2,3).
- The second and third characters identify the component that wrote the message. The major components are:
  CI Command-line interpreter
  CL Class loader
  LK Locking
  ST Storage (Memory management)
  TI Title
  XE Execution engine
- The remainder is a unique string, `JAVAVERSION` in the example above.
- Special tags have these characteristics:
  - A tag of `NULL` means the line is just to aid readability.
  - Every section is headed by a tag of `0SECTION` with the section title.

Here is an example of some tags taken from the start of a Javadump. The components are highlighted for clarification:

```
NULL        ------------------------------------------------------------------------
0SECTION    TITLE subcomponent dump routine
NULL        ===============================
1TISIGINFO  Dump Event "vmstop" (00000002) Detail "#00000000" received
1TIDATETIME Date: 2006/07/21 at 09:17:04
1TIFILENAME Javacore filename: /home/test/javacore.20060721.091704.12909.txt
NULL        ------------------------------------------------------------------------
0SECTION    GPINFO subcomponent dump routine
NULL        ===============================
2XHOSLEVEL  OS Level : Linux 2.6.14-ray8.1smp
2XHCPUS     Processors -
3XHCPUARCH    Architecture : x86
3XHNUMCPUS    How Many : 1
```

For the rest of the section, the tags are removed to aid readability.

Title, GPInfo, and EnvInfo sections

The first three sections of a Javadump are Title, GPInfo, and EnvInfo. This example shows some output taken from a simple Java test program running with WebSphere Real Time. This calls (using JNI) an external function that de-references an invalid pointer, which produces a “general protection fault” (GPF).

TITLE The `TITLE` section of the Javadump shows basic information about the event that caused the generation of the Javadump, the time it was taken, and its name.

GPINFO The GPINFO section varies in content depending on whether the Javadump was produced because of a GPF or not. It shows some general information about the operating system, including details of the Linux kernel level being used and the number of available processors. If the failure was caused by a GPF, GPF information about the failure is provided, in this case showing that the protection exception was thrown from libxdump_event_generator.so. The registers specific to the processor and architecture are also displayed.

ENVINFO
The `ENVINFO` section shows information about the JRE level that failed and details about the command line that launched the JVM process and the JVM environment in place.

```
NULL           ------------------------------------------------------------------------
0SECTION       TITLE subcomponent dump routine
NULL           ===============================
1TISIGINFO     Dump Event "gpf" (00002000) received
1TIDATETIME    Date:                2006/08/01 at 05:02:21
1TIFILENAME    Javacore filename:    /home/test/javacore.20060801.050221.16137.txt
NULL           ------------------------------------------------------------------------
0SECTION       GPINFO subcomponent dump routine
NULL           ===============================
2XHOSLEVEL     OS Level         : Linux 2.6.16-rtj12.5smp
2XHCPUS        Processors -
3XHCPUARCH       Architecture   : x86
3XHNUMCPUS       How Many       : 4
NULL
1XHEXCPCODE    J9Generic_Signal_Number: 00000004
1XHEXCPCODE    Signal_Number: 0000000B
1XHEXCPCODE    Error_Value: 00000000
1XHEXCPCODE    Signal_Code: 00000001
1XHEXCPCODE    Handler1: B7F27885
1XHEXCPCODE    Handler2: B7EEE786
1XHEXCPCODE    InaccessibleAddress: 00000000
NULL
1XHEXCPMODULE  Module: /home/test/native/bin/linux_x86-32/libxdump_event_generator.so
1XHEXCPMODULE  Module_base_address: B25B4000
1XHEXCPMODULE  Symbol: Java_ProvokeEvent_GenerateGPFEvent
1XHEXCPMODULE  Symbol_address: B25B84EB
NULL
1XHREGISTERS   Registers:
2XHREGISTER      EDI:085DE300
2XHREGISTER      ESI:B7F88020
2XHREGISTER      EAX:00000000
2XHREGISTER      EBX:B25C1370
2XHREGISTER      ECX:00000000
2XHREGISTER      EDX:00000001
2XHREGISTER      EIP:B25B852D
2XHREGISTER      ES:0000007B
2XHREGISTER      DS:0000007B
2XHREGISTER      ESP:B226406C
2XHREGISTER      EFlags:00210286
2XHREGISTER      CS:00000073
2XHREGISTER      SS:0000007B
2XHREGISTER      EBP:B2264074
NULL
1XHFLAGS       VM flags:00040000
NULL
NULL           ------------------------------------------------------------------------
0SECTION       ENVINFO subcomponent dump routine
NULL           ================================
1CIJAVAVERSION J2RE 5.0 IBM J9 2.3 Linux x86-32 build 20060731_07313_lHdRRr (JIT enabled - 20060728_1809_r8.rt)
1CIRUNNINGAS   Running as a standalone JVM
1CICMDLINE     java -Xrealtime -agentlib:xdump_event_generator XDumpEventGenerator gpf
1CIJAVAHOMEDIR Java Home Dir:   /home/test/sdk/jre
1CIJAVADLLDIR  Java DLL Dir:    /home/test/sdk/jre/bin
1CISYSCP       Sys Classpath:   /home/test/sdk/jre/bin/realtime/jclSC150/realtime.jar;/home/test/...
1CIUSERARGS    UserArgs:
2CIUSERARG               -Xjcl:jclscar_23
2CIUSERARG               -Dcom.ibm.oti.vm.bootstrap.library.path=/home/test/...
2CIUSERARG               -Dsun.boot.library.path=/home/test/sdk/jre/bin/realtime:/home/test/sdk/jre/bin
2CIUSERARG               -Djava.library.path=/home/test/sdk/jre/bin/realtime:/home/test/...
2CIUSERARG               -Djava.home=/home/test/sdk/jre
2CIUSERARG               -Djava.ext.dirs=/home/test/sdk/jre/lib/ext
2CIUSERARG               -Duser.dir=/home/test
2CIUSERARG               _j2se_j9 0xB7F82200
2CIUSERARG               vfprintf 0x0804B65C
2CIUSERARG               -Xrealtime
2CIUSERARG               -agentlib:xdump_event_generator
2CIUSERARG               -Dinvokedviajava
2CIUSERARG               -Djava.class.path=.
2CIUSERARG               vfprintf
2CIUSERARG               _port_library 0xB7F84560
2CIUSERARG               -Xdump
NULL
NULL           ---------------------- END OF DUMP -------------------------------------
```

Storage Management (MEMINFO)

An example of the MEMINFO section showing information from the memory manager component.

See the Diagnostics Guide for details about how the memory manager component works.

This part of the file gives various storage management values (in hexadecimal), including the free space and allocated sizes for the heap, the immortal memory area and any scoped memory areas being used by your application. There is one entry for each scope that the application creates (so in the example below, only one scope had been created). The file also gives details about other internal memory that the JVM is using. The example below shows some typical output.

```
NULL          ------------------------------------------------------------------------
0SECTION      MEMINFO subcomponent dump routine
NULL          ================================
1STHEAPFREE   Bytes of Heap Space Free: 3e8e800
1STHEAPALLOC  Bytes of Heap Space Allocated: 4000000
1STHEAPFREE   Bytes of Immortal Space Free: 0
1STHEAPALLOC  Bytes of Immortal Space Allocated: 1000000
1STHEAPFREE   Bytes of Scoped Space ID=00A6D0B8 Free: 2800
1STHEAPALLOC  Bytes of Scoped Space Allocated: 2800
NULL
1STSEGTYPE    Internal Memory
NULL          segment  start    alloc    end      type     bytes
1STSEGMENT    00CAFD20 00F33008 00F3FE6C 00F43008 01000040 10000
1STSEGMENT    00CAFC24 00F02E90 00F12E78 00F12E90 01000040 10000
<< lines removed for clarity >>

NULL
1STSEGTYPE    Object Memory
NULL          segment  start    alloc    end      type     bytes
1STSEGSUBTYPE Scoped Segment ID=00A6D0B8
1STSEGMENT    00CB0070 00F8E4F0 00F90CF0 00F90CF0 00002008 2800
1STSEGSUBTYPE Immortal Segment ID=00A6D084
1STSEGMENT    00CAFFF8 05180020 06180020 06180020 00001008 1000000
1STSEGSUBTYPE Heap Segment ID=00A6D068
1STSEGMENT    00CAFF80 01070020 05070020 05070020 00000009 4000000
NULL
1STSEGTYPE    Class Memory
NULL          segment  start    alloc    end      type     bytes
1STSEGMENT    00DD75B8 01039030 0104AE38 01059030 00020040 20000
1STSEGMENT    00DD7564 01031020 010314B8 01039020 00010040 8004
<< lines removed for clarity >>
NULL
1STSEGTYPE    JIT Code Cache
NULL          segment  start    alloc    end      type     bytes
1STSEGMENT    00CF41A8 06190000 06190000 06210000 00000068 80000
NULL
1STSEGTYPE    JIT Data Cache
NULL          segment  start    alloc    end      type     bytes
1STSEGMENT    00CF42E0 06210020 06212A0C 06290020 00000048 80000
```

Note:

- Javadumps produced by the standard JVM (without the **-Xrealtime** flag) contain a "GC History" section. This information does not appear in Javadumps produced by WebSphere Real Time. When using WebSphere Real Time, you should use the **-verbose:gc** option or the JVM snap trace to obtain information about GC behavior. See "Using the -verbose:gc option" on page 23 and "Obtaining JVM trace information" on page 114 for more details of these tools.
- If you are running a program which uses scoped memory, you may find that if an OutOfMemoryError is thrown, none of the memory areas listed in the Javadump appear to be empty. This is because when a scope which is nested within another scope runs out of memory, by the time the Javadump is generated, the inner scope may have been deleted. To get

information which relates to the state of the memory areas at the time the OutOfMemoryError is thrown, re-run your program with the following command-line option:

```
-Xdump:java:events=throw,filter=java/lang/OutOfMemoryError,range=1..1
```

This generates a Javadump (additional to any that are later generated by default dump agents) when the OutOfMemoryError is thrown, rather than when the uncaught exception is detected (which happens slightly later). Within this Javadump you should be able to see all memory areas which were active at the time the OutOfMemoryError was thrown, including any inner scopes. For further information on using the **-Xdump** option, please refer to "Using dump and trace agents" on page 111.

Locks, monitors, and deadlocks (LOCKS)

An example of the LOCKS component part of a Javadump taken during a deadlock. A lock (also referred to as a monitor) prevents more than one entity from accessing a shared resource. Each object in Java has an associated lock (obtained by using a synchronized block or method). In the case of the JVM, threads compete for various resources in the JVM and locks on Java objects.

This example was taken from a simple deadlock test program where two threads "DeadLockThread 0" and "DeadLockThread 1" were attempting to synchronize (Java keyword) on two java/lang/Integers and were not compatible.

You can see in the example (highlighted) that "DeadLockThread 1" has locked the object instance java/lang/Integer@004B2290. The monitor has been created as a result of a Java code fragment looking like "synchronize(count0)", and this monitor has "DeadLockThread 1" waiting to get a lock on this same object instance (count0 from the code fragment). Below the highlighted section is another monitor locked by "DeadLockThread 0" that has "DeadLockThread 1" waiting.

This classic deadlock situation is caused by an error in application design; Javadump is a major tool in the detection of such events.

```
------------------------------------------------------------------------
LOCKS subcomponent dump routine
===============================

   Monitor pool info:
    Current total number of monitors: 2

Monitor Pool Dump (flat & inflated object-monitors):
      sys_mon_t:0x00039B40 infl_mon_t: 0x00039B80:
       java/lang/Integer@004B22A0/004B22AC: Flat locked by "DeadLockThread 1"
                                            (0x41DAB100), entry count 1
            Waiting to enter:
                "DeadLockThread 0" (0x41DAAD00)      sys_mon_t:0x00039B98 infl_mon_t: 0x00039BD8:
      java/lang/Integer@004B2290/004B229C: Flat locked by "DeadLockThread 0"
                                                        (0x41DAAD00), entry count 1
            Waiting to enter:
                "DeadLockThread 1" (0x41DAB100)
JVM System Monitor Dump (registered monitors):
          Thread global lock (0x00034878): <unowned>
          NLS hash table lock (0x00034928): <unowned>
          portLibrary_j9sig_async_monitor lock (0x00034980): <unowned>
          Hook Interface lock (0x000349D8): <unowned>
            < lines removed for brevity >

======================
Deadlock detected !!!
---------------------
```

```
  Thread "DeadLockThread 1" (0x41DAB100)
    is waiting for:
      sys_mon_t:0x00039B98 infl_mon_t: 0x00039BD8:
      java/lang/Integer@004B2290/004B229C:
    which is owned by:
  Thread "DeadLockThread 0" (0x41DAAD00)
    which is waiting for:
      sys_mon_t:0x00039B40 infl_mon_t: 0x00039B80:
      java/lang/Integer@004B22A0/004B22AC:
    which is owned by:
  Thread "DeadLockThread 1" (0x41DAB100)
```

Threads and stack trace (THREADS)

For the application programmer, one of the most useful pieces of a Javadump is the THREADS section. This section shows a complete list of Java™ threads, including real-time threads and no-heap real-time threads, that are alive.

Note: A Javadump that is produced from a no-heap real-time thread may have some missing information. For threads where the thread name object is not visible from the no-heap real-time thread, the text "(access error)" is printed instead of the actual thread name.

A thread is alive if it has been started but not yet stopped. An application-level thread is implemented by a native thread of the operating system. Each thread in a Javadump from WebSphere Real Time is represented by a line such as:

```
"Signal Dispatcher" TID:0x41509200, j9thread_t:0x0003659C, state:R,prio=5
       (native thread ID:5820, native priority:0, native policy:SCHED_OTHER)
    at com/ibm/misc/SignalDispatcher.waitForSignal(Native Method)
    at com/ibm/misc/SignalDispatcher.run(SignalDispatcher.java:84)
```

The properties on the first line are thread name, identifier, JVM data structure address, current state and Java priority, and on the second line the native operating system thread ID, native operating system thread priority and native operating system scheduling policy.

Each thread priority is mapped to an operating system priority value. For further details of how this works for Java threads, real-time threads and no-heap real-time threads, see "Priority mapping" on page 31.

The values of state can be:

- R - Runnable - the thread is able to run when given the chance.
- CW - Condition Wait - the thread is waiting. For example, because:
  - A sleep() call is made
  - The thread has been blocked for I/O
  - A synchronized method of an object locked by another thread has been called
  - The thread is synchronizing with another thread with a join() call
- S - Suspended - the thread has been suspended by another thread.
- Z - Zombie - the thread has been killed.
- P - Parked - the thread has been parked by the new concurrency API (java.util.concurrent).
- B - Blocked - the thread is waiting to obtain a lock that something else currently owns.

There is a stack trace below each Java thread. A stack trace is a representation of the hierarchy of Java method calls made by the thread. The example below is taken

from the same Javadump as used in the LOCKS example and two threads ("DeadLockThread 0" and "DeadLockThread 1") are both in blocked state. The application code path that resulted in the deadlock between "DeadLockThread 0" and "DeadLockThread 1" can clearly be seen.

```
------------------------------------------------------------------------
THREADS subcomponent dump routine
================================

Current Thread Details
----------------------

All Thread Details
------------------

Full thread dump J9SE VM (J2RE 5.0 IBM J9 2.3 Linux x86-32 build 20060714_07194_lHdSMR,
native threads):
    "DestroyJavaVM helper thread" TID:0x41508A00, j9thread_t:0x00035EAC, state:CW, prio=5
           (native thread ID:3924, native priority:0, native policy:SCHED_OTHER, scope:00A6D068)
    "JIT Compilation Thread" TID:0x41508E00, j9thread_t:0x000360FC, state:CW, prio=11
           (native thread ID:188, native priority:11, native policy:SCHED_OTHER, scope:00A6D068)
    "Signal Dispatcher" TID:0x41509200, j9thread_t:0x0003659C, state:R, prio=5
           (native thread ID:3192, native priority:0, native policy:SCHED_OTHER, scope:00A6D084)
        at com/ibm/misc/SignalDispatcher.waitForSignal(Native Method)
        at com/ibm/misc/SignalDispatcher.run(SignalDispatcher.java:84)
    "DeadLockThread 0" TID:0x41DAAD00, j9thread_t:0x42238A1C, state:B, prio=5
           (native thread ID:1852, native priority:0, native policy:SCHED_OTHER, scope:00A6D068)
        at Test$DeadlockThread0.SyncMethod(Test.java:112)
        at Test$DeadlockThread0.run(Test.java:131)
    "DeadLockThread 1" TID:0x41DAB100, j9thread_t:0x42238C6C, state:B, prio=5
           (native thread ID:1848, native priority:0, native policy:SCHED_OTHER, scope:00A6D068)
        at Test$DeadlockThread1.SyncMethod(Test.java:160)
        at Test$DeadlockThread1.run(Test.java:141)
```

Classloaders and Classes (CLASSES)

An example of the classloader (CLASSES) section that includes Classloader summaries and Classloader loaded classes. Classloader summaries are the defined class loaders and the relationship between them. Classloader loaded classes are the classes that are loaded by each classloader.

See the Diagnostics Guide for information about the parent-delegation model.

In this example, there are the standard three classloaders:

- Application classloader (sun/misc/Launcher$AppClassLoader), which is a child of the extension classloader.
- The Extension classloader (sun/misc/Launcher$ExtClassLoader), which is a child of the bootstrap classloader.
- The Bootstrap classloader. Also known as the System classloader.

The example below shows this relationship. Take the application classloader with the full name sun/misc/Launcher$AppClassLoader. Under `Classloader summaries`, it has flags `-----ta-`, which show that the class loader is t=trusted and a=application. It gives the number of loaded classes (1) and the parent classloader as sun/misc/Launcher$ExtClassLoader.

Under the `ClassLoader loaded classes` heading, you can see that the application classloader has loaded three classes, one called Test at address 0x41E6CFE0.

In this example, the System class loader has loaded a large number of classes, which provide the basic set from which all applications derive.

```
------------------------------------------------------------------------
CLASSES subcomponent dump routine
================================
Classloader summaries
        12345678: 1=primordial,2=extension,3=shareable,4=middleware,
                  5=system,6=trusted,7=application,8=delegating
        p---st--        Loader *System*(0x00439130)
                        Number of loaded classes 306
        -x--st--        Loader sun/misc/Launcher$ExtClassLoader(0x004799E8),
                        Parent *none*(0x00000000)
                        Number of loaded classes 0
        -----ta-        Loader sun/misc/Launcher$AppClassLoader(0x00484AD8),
                        Parent sun/misc/Launcher$ExtClassLoader(0x004799E8)
                        Number of loaded classes 1
ClassLoader loaded classes
Loader *System*(0x00439130)
  java/security/CodeSource(0x41DA00A8)
  java/security/PermissionCollection(0x41DA0690)
    << 301 classes removed for clarity >>
  java/util/AbstractMap(0x4155A8C0)
  java/io/OutputStream(0x4155ACB8)
  java/io/FilterOutputStream(0x4155AE70)
Loader sun/misc/Launcher$ExtClassLoader(0x004799E8)
Loader sun/misc/Launcher$AppClassLoader(0x00484AD8)
  Test(0x41E6CFE0)
  Test$DeadlockThread0(0x41E6D410)
  Test$DeadlockThread1(0x41E6D6E0)
```

Environment variables and Javadump

Although the preferred mechanism of controlling the production of Javadumps is now by the use of dump agents using **-Xdump:java**, you can also use the previous mechanism, environment variables.

The table below details environment variables specifically concerned with Javadump production.

Table 20. Environment variables used to produce Javadumps

Environment Variable	Usage Information
DISABLE_JAVADUMP	Setting **DISABLE_JAVADUMP** to true is the equivalent of using **-Xdump:java:none** and stops the default production of javadumps.
IBM_JAVACOREDIR	The default location into which the Javacore will be written. See "Location of the generated Javadump" on page 121.
JAVA_DUMP_OPTS	Use this environment variable to control the conditions under which Javadumps (and other dumps) are produced. See "Using core (system) dumps" on page 134 for more information.
IBM_JAVADUMP_OUTOFMEMORY	By setting this environment variable to false, you disable Javadumps for an OutOfMemory condition.

Using Heapdump

The term Heapdump describes the IBM Virtual Machine for Java mechanism that generates a dump of all the live objects that are in the Java heap, Immortal and Scoped Memory Areas; that is, those that are being used by the running Java application. This dump is stored in a file (using the phd format). You can use various tools on the Heapdump output to analyze the composition of the objects on the heap and (for example) help to find the objects that are controlling large amounts of memory on the Java heap and the reason why the Garbage Collector cannot collect them.

This section describes:

Getting Heapdumps

A Heapdump can be generated from a running JVM by either explicit generation or JVM-triggered generation.

Using the default dump agents provided, when the Java heap is exhausted that is, the OutOfMemoryError condition is encountered and the resulting exception is not caught or handled by the application, JVM-triggered generation of a Heapdump occurs. See "Using dump and trace agents" on page 111 for more information.

Heapdumps can be generated in other situations by use of **-Xdump:heap** to generate dumps based on specific events, for example, a user-generated signal, or programmatically by use of the com.ibm.jvm.Dump.Heapdump() method from within application code.

Heapdumps, by default are a single PHD formatted file containing information about all the objects within the JVM's memory areas, Heap, Immortal and Scoped memory. WebSphere Real Time can also be configured to produce individual heapdumps for each memory area or to produce text formatted (classic) heapdumps. See "Enabling multiple Heapdumps for real-time JVMs" for real-time JVMs and "Enabling text formatted ("classic") Heapdumps" on page 131 for more details.

To display on JVM startup the conditions (if any) that will generate a Heapdump (or javadump or systemdump), you can use **-Xdump:what**. See "Using dump and trace agents" on page 111 for more information.

Environment variables can also affect the generation of Heapdumps (although this is a deprecated mechanism). See "Environment variables and Heapdump" on page 132 for more details.

Enabling multiple Heapdumps for real-time JVMs

The generated Heapdump is by default a single file containing information about all Java objects in all memory areas, Heap memory, Immortal memory and Scoped memory. The main reason to produce multiple dumps is so that each individual heap area can be analyzed using the traditional heap dump tools without modification.

You can also obtain separate Heapdumps containing information about Java objects within each memory area by using the **request=multiple** option with **-Xdump:heap**. This produces a set of Heapdumps with an extra field in the name indicating the specific memory area:

```
heapdump.%id.%Y%m%d.%H%M%S.%pid.phd
```

where *%id* identifies the Heapdump file as containing objects in Heap memory, Immortal memory or a specific area of Scoped memory.

There are 4 types of heap represented by the following names: “Default”, “Immortal”, “Scope” and “Other”. The heapdump code replaces the %id in the heap label with one of these names concatenated with an identifier (typically numeric), for example: `heapdump.Immortal12994208.20060807.093653.7684.txt.`

Example

```
java -Xrealtime -Xdump:heap:request=multiple,opts=CLASSIC <java program>
```

Using this extra option produces multiple heapdumps in the CLASSIC text format.

```
java -Xrealtime -Xdump:heap:request=multiple <java program>
```

Using this extra option produces multiple heapdumps in the portable heapdump (phd) format.

If you specify file= or label=, this changes the filename template that is used to derive the eventual heapdump filename, for example:

```
java -Xrealtime -Xdump:heap:file=dump.%id.phd,request=multiple <java program>
```

Using this extra option produces dumps with the name `dump.<heap type>.phd`

Enabling text formatted (″classic″) Heapdumps

The generated Heapdump is by default in the binary, platform-independent, phd format, which can be examined using the available tooling.

The generated Heapdump is by default in the binary, platform-independent, phd format, which can be examined using the available tooling. See “Available tools for processing Heapdumps” on page 132. However, it is sometimes useful to have an immediately readable view of the heap. You can obtain this view by using the **opts=CLASSIC** or **opts=PHD+CLASSIC** (**opts=PHD** is the default) with **-Xdump:heap** (see “Using dump and trace agents” on page 111) or by defining one of the following environment variables:

- **IBM_JAVA_HEAPDUMP_TEST**, which allows you to perform the equivalent of **opts=PHD+CLASSIC**
- **IBM_JAVA_HEAPDUMP_TEXT**, which allows the equivalent of **opts=CLASSIC**

Location of the generated Heapdump

The JVM checks a these locations for existence and write-permission, then stores the Heapdump in the first one that is available.

- The location that is specified using the **file** suboption on the triggered **-Xdump:heap** agent.
- The location that is specified by the **IBM_HEAPDUMPDIR** environment variable, if set.
- The current working directory of the JVM processes.
- The location that is specified by the **TMPDIR** environment variable, if set.

- The /tmp directory.

The size of the Heapdump file depends on the size of the Java heap. The phd heapdump format is compressed. If ″Classic″ Heapdump is enabled, you will need more space to store the dump.

The generated Heapdump will have a name of the form:

```
heapdump.%Y%m%d.%H%M%S.%pid.phd
```

if the request=multiple -Xdump option is used the Heapdumps will have a name of the form:

```
heapdump.%id.%Y%m%d.%H%M%S.%pid.phd
```

where:
- %id identifies the memory area the heapdump represents. The id begins with the memory area name either Default for Java heap or Immortal or Scoped.
- `%pid` is the process ID
- `.%Y%m%d.%H%M%S` is the date and time

Note:

1. If ″Classic″ Heapdump is enabled, the name of the Heapdump will end in txt rather than phd.
2. You can override the standard names by use of the **label=** parameter. See “Using dump and trace agents” on page 111 for more information.

Available tools for processing Heapdumps

There are several tools available for Heapdump analysis through IBM support Web sites.

The preferred Heapdump analysis tool Memory Dump Diagnostic for Java (MDD4J) is also available in IBM Support Assistant: http://www-306.ibm.com/software/support/isa/

Further details of the range of available tools can be found at http://www.ibm.com/support/docview.wss?uid=swg24009436

Using verbose:gc to obtain heap information

Use the verbose:gc utility to obtain information about the Java Object heap in real time while running your Java applications.

To activate this utility, run Java with the **-verbose:gc** option:

```
java -Xrealtime -verbose:gc
```

For more information, see “Using the -verbose:gc option” on page 23.

Environment variables and Heapdump

Although the preferred mechanism for controlling the production of Heapdumps is now the use of dump agents with **-Xdump:heap**, you can also use the previous mechanism, environment variables.

Table 21 on page 133 details environment variables specifically concerned with Heapdump production.

Table 21. Environment variables used to produce heap dumps

Environment Variable	Usage Information
IBM_HEAPDUMPIBM_HEAP_DUMP	Setting either of these to any value (such as true) enables heap dump production by means of signals.
IBM_HEAPDUMPDIR	The default location into which the Heapdump will be written. See "Location of the generated Heapdump" on page 131.
JAVA_DUMP_OPTS	Use this environment variable to control the conditions under which Heapdumps (and other dumps) are produced. See "Using core (system) dumps" on page 134 for more information .
IBM_HEAPDUMP_OUTOFMEMORY	By setting this environment variable to false, you disable Heapdumps for an OutOfMemory condition.
IBM_JAVA_HEAPDUMP_TEST	Use this environment variable to cause the JVM to generate both phd and text versions of Heapdumps. Equivalent to **opts=PHD+CLASSIC** on the **-Xdump:heap** stanza.
IBM_JAVA_HEAPDUMP_TEXT	Use this environment variable to cause the JVM to generate a text (human readable) Heapdump. Equivalent to **opts=CLASSIC** on the **-Xdump:heap** stanza.

Using core (system) dumps

The JVM can generate system dumps, also known as core dumps in response to specific events. Core dumps are platform-specific and normally quite large. The tools used to analyze core dumps are also platform-specific. For example, gdb on Linux.

Dump agents are the primary method for controlling the generation of core dump. See "Using dump and trace agents" on page 111 for more information on dump agents.

"Using the dump viewer" on page 140 describes a **jextract** and **jdmpview -Xrealtime** that is used to analyze core dumps at the JVM level.

To maintain backwards compatibility, the JVM supports the use of environment variables for core dump initiation. See "Environment variables and core dumps" on page 135 for more information.

This section contains the following sections:

- "Defaults"
- "Location of the generated core dump"
- "Environment variables and core dumps" on page 135

Defaults

This topic describes the default agents for producing core dumps when using the JVM.

Using the **-Xdump:what** option shows the following core (system) dump agent:

```
dumpFn=doSystemDump
events=gpf+abort
filter=
label=/home/test/core.%Y%m%d.%H%M%S.%pid.dmp
range=1..0
priority=999
request=serial
opts=
```

This output shows that by default a core dump is produced in these cases:

- A general protection fault occurs. (For example, branching to memory location 0, or a protection exception.)
- An abort is encountered. (For example, native code has called abort() or when using `kill -ABRT` on UNIX

Location of the generated core dump

When a core dump is triggered, the JVM checks each of the following locations for existence and write-permission, and stores the core dump in the first one available.

- The location specified using the **file** suboption on the triggered **-Xdump:system** agent. See "Using dump and trace agents" on page 111.
- The location specified by the **IBM_COREDIR** 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 .

Enough free disk space must be available for the System dump file to be written correctly. On Linux platforms the system dump will be written to the default OS specified directory (typically the current working directory) before being moved to the final destination. The default directory must have enough free disk space and appropriate permissions.

The file name is of the following form: `core.%Y%m%d.%H%M%S.%pid.dmp`

where
- %pid is the process ID
- %Y%m%d is the year month and day.
- %H%M%S is the time.

Environment variables and core dumps

The **-Xdump** option on the command line is the preferred method for generating core dumps for cases where the default settings are not enough.

However, the deprecated **JAVA_DUMP_OPTS** environment variable has been retained for compatibility for producing core dumps. If you set the **JAVA_DUMP_OPTS** environment variable, default dump agents will be disabled, however any **-Xdump** options specified on the command line are used.

The **JAVA_DUMP_OPTS** environment variable is used as follows:

```
JAVA_DUMP_OPTS="ONcondition(dumptype,dumptype),ONcondition(dumptype,...),...)
```

where:
- *condition* can be:
  - ANYSIGNAL
  - DUMP
  - ERROR
  - INTERRUPT
  - EXCEPTION
  - OUTOFMEMORY
- and *dumptype* can be:
  - ALL
  - NONE
  - JAVADUMP
  - SYSDUMP
  - HEAPDUMP

JAVA_DUMP_OPTS is parsed by taking the first (leftmost) occurrence of each condition, so duplicates are ignored. That is,

```
ONERROR(SYSDUMP),ONERROR(JAVADUMP)
```

creates system dumps for error conditions. Also, the `ONANYSIGNAL` condition is parsed before all others, so

```
ONINTERRUPT(NONE),ONANYSIGNAL(SYSDUMP)
```

has the same effect as

```
ONANYSIGNAL(SYSDUMP),ONINTERRUPT(NONE)
```

If the **JAVA_DUMP_TOOL** environment variable is set, that variable is assumed to specify a valid executable name and is parsed for replaceable fields, such as %pid. If %pid is detected in the string, the string is replaced with the JVM's own process ID. The tool specified by **JAVA_DUMP_TOOL** is run after any system or heap dump has been taken, before anything else.

If the **OUTOFMEMORY** condition is used, it overrides the **IBM_HEAPDUMP_OUTOFMEMORY** and **IBM_JAVADUMP_OUTOFMEMORY** settings and takes the prescribed dumps whenever an out-of-memory exception is thrown (even if it is handled).

Using method trace

Using method trace provides a complete (and potentially large) diagnosis of code paths inside your application and also inside the system classes. Method trace is a powerful and free tool that allows you to trace methods in any Java code. You do not have to add any hooks or calls to existing code. Use wild cards and filtering to control method trace to focus on the sections of code that interest you.

Method trace can trace:
- Method entry
- Method exit

Use method trace to debug and trace application code and the system classes provided with the JVM.

While method trace is powerful, it also has a cost. Application throughput will be significantly impacted by method trace, proportionally to the number of methods traced. Additionally, trace output is reasonably large and can grow to consume a significant amount of drive space. For instance, full method trace of a "Hello World" application is over 10 MB.

Method trace is part of the larger 'JVM trace' package. JVM trace is described in the Diagnostics Guide.

This section describes the basic use of trace. When you feel comfortable using trace, see the Diagnostics Guide for more detailed information

Running with method trace

You can control method trace by using the command-line option **-Xtrace:<option>**.

If you want method trace to be formatted, set two trace options:
- **-Xtrace:methods** — set this option to decide what to trace.
- **-Xtrace:print** — set this option to 'mt' to invoke method trace.

The first property is only a constant: `-Xtrace:print=mt`

Use the `methods` parameter to control what is traced. To trace everything, set it to `methods=*.*`. This is not recommended because you are certain to be overwhelmed by the amount of output.

The `methods` parameter is formally defined as follows:

```
-Xtrace:methods=[[!]method_spec[,...]]
```

Where `method_spec` is formally defined as:

```
{*|[*]classname[*]}.{*|[*]methodname[*]}[()]
```

Note that
- The delimiter between parts of the package name is a forward slash, '/', even on platforms like Windows that use a backward slash as a path delimiter.
- The "!" in the methods parameter is a NOT operator that allows you to tell the JVM not to trace the specified method or methods. Use this with other `methods` parameters to set up a trace of the form: "trace methods of this type but not methods of that type".

- The parentheses, (), that are in the method_spec define whether or not to trace method parameters.
- If a method specification includes any commas, the whole specification must be enclosed in braces, for example:

  ```
  -Xtrace:methods={java/lang/*,java/util/*},print=mt
  ```
- It might be necessary to enclose your command line in quotation marks to prevent the shell intercepting and fragmenting comma-separated command lines, for example:

  ```
  "-Xtrace:methods={java/lang/*,java/util/*},print=mt"
  ```

Examples

Tracing entry and exit of all methods in a given class:

```
"-Xtrace:{methods=ReaderMain.*,java/lang/String.*},iprint=mt"
```

This traces all method entry and exit of the ReaderMain class in the default package and the java.lang.String class.

Tracing entry, exit and input parameters of all methods in a class:

```
"-Xtrace:methods=ReaderMain.*(),iprint=mt"
```

This traces all method entry, exit, and input of the ReaderMain class in the default package.

Tracing all methods in a given package:

```
"-Xtrace:methods=com/ibm/socket/*.*(),iprint=mt"
```

This traces all method entry, exit, and input of all classes in the package com.ibm.socket.

Multiple method trace:

```
"-Xtrace:methods={Widget.*(),common/*},iprint=mt"
```

This traces all method entry, exit, and input in the Widget class in the default package and all method entry and exit in the common package.

Using the ! operator

```
"-Xtrace:methods={ArticleUI.*,!ArticleUI.get*},iprint=mt"
```

This traces all methods in the ArticleUI class in the default package except those beginning with “get”.

Where does the output appear?

In this simple case, output appears on the ’stderr’. If you want to store your output, redirect this stream to a file.

You can also write method trace to a file directly, as described in in the *Advanced options*Diagnostics Guide.

Example of method trace

Using the **-Xtrace** option.

```
java "-Xtrace:methods=java/lang*.*,iprint=mt" HW
```

Results:

```
10:02:42.281*0x9e900      mt.4         > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.281 0x9e900      mt.4          > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
```

```
                         V Compiled static method
10:02:42.281 0x9e900      mt.4           > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                         V Compiled static method
10:02:42.281 0x9e900      mt.4            > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                         V Compiled static method
10:02:42.281 0x9e900      mt.10           < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                         V Compiled static method
10:02:42.281 0x9e900      mt.10          < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                         V Compiled static method
10:02:42.281 0x9e900      mt.4           > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                         V Compiled static method
10:02:42.281 0x9e900      mt.10          < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                         V Compiled static method
10:02:42.281 0x9e900      mt.10         < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                         V Compiled static method
10:02:42.281 0x9e900      mt.4          > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                         V Compiled static method
10:02:42.281 0x9e900      mt.4            > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                         V Compiled static method
10:02:42.296 0x9e900      mt.10           < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                         V Compiled static method
10:02:42.296 0x9e900      mt.10         < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                         V Compiled static method
10:02:42.296 0x9e900      mt.4          > java/lang/String.<clinit>()V Compiled static method
10:02:42.296 0x9e900      mt.4           > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                         V Compiled static method
10:02:42.296 0x9e900      mt.4            > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                         V Compiled static method
10:02:42.296 0x9e900      mt.4             > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                         V Compiled static method
10:02:42.296 0x9e900      mt.10            < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                         V Compiled static method
10:02:42.296 0x9e900      mt.4             > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                         V Compiled static method
10:02:42.328 0x9e900      mt.10            < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                         V Compiled static method
10:02:42.328 0x9e900      mt.10           < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                         V Compiled static method
10:02:42.328 0x9e900      mt.4            > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                         V Compiled static method
10:02:42.328 0x9e900      mt.10           < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                         V Compiled static method
10:02:42.328 0x9e900      mt.4            > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                         V Compiled static method
10:02:42.328 0x9e900      mt.10           < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                         V Compiled static method
10:02:42.328 0x9e900      mt.10          < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                         V Compiled static method
```

The output lines comprise:

- `0x9e900`, the current JVM thread identifier. All trace with the same JVM thread identifier relates to a single thread.
- The individual tracepoint id in the mt component that actually collects and emits the data.
- The remaining fields show whether a method is being entered (>) or exited (<), followed by details of the method.

Using the dump viewer

System dumps are produced in a platform-specific binary format, typically as a raw memory image of the process that was running at the time the dump was initiated. The SDK dump viewer allows you to navigate around the dump, and obtain information in a readable form, with symbolic (source code) data where this can be derived.

You can view Java information (for example, threads and objects on the heap) and native information (for example, native stacks, libraries, and raw memory locations). You can run the dump viewer on one platform to work with dumps from another platform. For example, you can look at Linux dumps on a Windows platform.

Note: **jdmpview -Xrealtime** is a new and different tool from the standard JVM jdmpview. **jdmpview -Xrealtime** is based on an application called DTFJView. The main difference is that DTFJView can handle information on multiple heaps and uses a different command set. In the output of **jdmpview -Xrealtime** there are references to DTFJView and these refer to **jdmpview -Xrealtime**.

Dump extractor: jextract

To use the full facilities of the dump viewer you must first run the **jextract** tool on the system dump. The **jextract** tool obtains platform specific information such as word size, endianness, data structure layouts, and symbolic information. It puts this information into an XML file. The **jextract** tool must be run on the same platform and the same JVM level (ideally the same machine) that was being used when the dump was produced. The combination of the dump file and the XML file produced by **jextract** allows the dump viewer (**jdmpview**) to analyze and display Java information.

jextract is in the directory sdk/jre/bin.

To invoke **jextract**, from a shell prompt, enter:

```
jextract -Xrealtime <core_file> [<zip_file>]
```

or

```
jextract -Xrealtime -nozip <core_file> [<xml_file>]
```

The **jextract** tool accepts these parameters:

-nozip
Do not compress the output data.

By default, output is written to a file called <core_file>.zip in the current directory. This file is a compressed file that contains:

- The dump
- xml produced from the dump, containing details of useful internal JVM information
- Other files that can help in diagnosing the dump (such as trace entry definition files)

Normally, you would send the jar file to IBM for problem diagnosis. The Diagnostics Guide tells you how to do that.

To analyze the dump locally, run **jdmpview -Xrealtime** directly on the zip:

```
jdmpview -Xrealtime -zip <core_file>.zip
```

If you run **jextract** on a JVM level that is different from the one for which the dump was produced you will see the following messages:

```
J9RAS.buildID is incorrect (found e8801ed67d21c6be, expecting eb4173107d21c673).
This version of jextract is incompatible with this dump.
Failure detected during jextract, see previous message(s).
```

This error message also occurs if you use **jextract** without the **-Xrealtime** flag when you are processing a core that was produced with the **-Xrealtime** flag. Similarly you must not use the **-Xrealtime** flag with **jextract** when processing a core that was produced from the standard JVM.

You can still use the dump viewer on the dump, but it is limited in the detail that it can show.

The contents of the zip file produced and the contents of the xml are subject to change, you are advised not to design tools based on the contents of these.

Dump viewer: jdmpview -Xrealtime

The dump viewer is a cross-platform tool that allows you to examine the contents of system dumps produced from the JVM. To be able to analyze platform-specific dumps, the dump viewer can use metadata created by the **jextract** tool. The dump viewer allows you to view both Java and native information from the time the dump was produced.

jdmpview is in the directory sdk/bin.

To invoke **jdmpview -Xrealtime**, from a shell prompt, enter:

```
jdmpview -Xrealtime -zip <zip_file>
```

or

```
jdmpview -Xrealtime -core <core_file>
```

The **jdmpview -Xrealtime** tool accepts these parameters:

-core *<core_file>*
Specify a dump file.

-zip *<zip_file>*
Specify a zip file containing the core file and associated XML file (produced by jextract).

Typical usage is `jdmpview -Xrealtime -core <core_file>`. The **jdmpview** tool opens and verifies the dump file and the associated xml file (*<core_file>*.xml).

After **jdmpview -Xrealtime** processes the arguments with which it was launched, it displays the message

```
For a list of commands, type "help"; for how to use "help", type "help help"
>
```

When you see this message, you can start invoking commands on **jdmpview**.

You can significantly improve the performance of **jdmpview** against larger dumps by ensuring that your system has enough memory available to avoid paging. On

larger dumps (that is, ones with large numbers of objects on the heap), you might have to invoke **jdmpview** using the **-Xmx** option to increase the maximum heap available to **jdmpview**:

```
jdmpview -Xrealtime -J-Xmx<n>
```

To pass command-line arguments to the JVM, you must prefix them with **-J**. For more information on using **-Xmx**, see the Diagnostics Guide.

Problems to tackle with the dump viewer

Dumps of JVM processes can arise either when you specify the **-Xdump** option on the command line or when the JVM is not in control (such as user-initiated dumps).

The extent to which jextract can analyze the information in a dump is affected by the state of the JVM when it was taken.

jdmpview is most useful in diagnosing customer-type problems and problems with the J2SE class libraries. A typical scenario is OutOfMemoryErrors in customer applications.

For problems involving gpfs, ABENDS, SIGSEVs, and similar problems, you will obtain more information by using the system debugger (gdb) with the dump file. The syntax for the gdb command is, `gdb <full_java_path> <system_dump_file>`. For example:

```
gdb /sdk/jre/bin/java core.20060808.173312.9702.dmp
```

However, jdmpview can still provide useful information when used alone. Because jdmpview allows you to observe stacks and objects, the tool enables introspection into a Java program much like a Java debugger would. It allows you to examine objects, follow reference chains and observe Java stack contents. The main difference (other than the user interface) is that the program state is frozen; thus no stepping can occur. However, this allows you to take periodic program snapshots and perform analysis to see what is happening at different times.

Commands for use with jdmpview -Xrealtime

jdmpview -Xrealtime is an interactive, command-line tool to explore the information from a JVM system dump in WebSphere Real Time and perform various complex analysis functions.

help [<*command_name*>**]**
Displays a list of commands or help for a specific command. With no parameters, **help** displays the complete list of commands currently supported. When a <*command_name*> is specified, **help** lists that command's sub-commands if it has sub-commands; otherwise, the command's complete description is displayed.

info thread [* | <*thread_name*>**]**
Displays information about Java and native threads. The command prints the following information about the current thread (no arguments), all threads ("*"), or the specified thread, respectively:
- Thread id
- Registers
- Stack sections
- Thread frames: procedure name and base pointer
- Associated Java thread (if applicable):
  - Name of Java thread

- Address of associated java.lang.Thread object
- Current state according to JVMTI specification
- Current state relative to java.lang.Thread.State
- The monitor the thread is waiting to enter or waiting on notify
- Thread frames: base pointer, method, and filename:line

info system
Displays the following information about the system the core dump:
- amount of memory
- operating system
- virtual machine(s) present

info class [<*class_name*>**]**
Displays the inheritance chain and other data for a given class. If no parameters are passed to **info class**, it prints the number of instances of each class and the total size of all instances of each class as well as the total number of instances of all classes and the total size of all objects. if a class name is passed to **info class**, it prints the following information about that class:
- name
- ID
- superclass ID
- class loader ID
- modifiers
- number of instances and total size of instances
- inheritance chain
- fields with modifiers (and values for static fields)
- methods with modifiers

info proc
Displays threads, command line arguments, environment variables, and shared modules of current process.

Note: To view the shared modules used by a process, use the **info sym** command.

info jitm
Displays JITed methods and their addresses:
- Method name and signature
- Method start address
- Method end address

info ls
Displays a list of available monitors and locked objects

info sym
Displays a list of available modules. For each process in the address spaces, this command outputs a list of module sections for each module with their start and end addresses, names, and sizes.

info mmap
Displays a list of all memory segments in the address space: start address and size.

info heap [* | <*heap_name*>**]**
Using no arguments displays the heap names and heap sections.

Using either "*" or a heap name displays the following information about all heaps or the specified heap:
- heap name
- (heap size and occupancy)

- heap sections
  - section name
  - section size
  - whether the section is shared
  - whether the section is executable
  - whether the section is read only

hexdump <*hex_address*> <*bytes_to_print*>
Displays a section of memory in a hexdump-like format. Displays <*bytes_to_print*> bytes of memory contents starting from <*hex_address*>.

+ Displays the next section of memory in hexdump-like format. This command is used in conjunction with the hexdump command to allow easy scrolling forwards through memory. It repeats the previous hexdump command starting from the end of the previous one.

- Displays the previous section of memory in hexdump-like format. This command is used in conjunction with the hexdump command to allow easy scrolling backwards through memory. It repeats the previous hexdump command starting from a position before the previous one.

whatis <*hex_address*>
Displays information about what is stored at the given memory address, <*hex_address*>. This command examines the memory location at <*hex_address*> and tries to find out more information about this address. For example, whether it is within an object in a heap or within the byte codes associated with a class method.

find
<*pattern*>,<*start_address*>,<*end_address*>,<*memory_boundary*>,<*bytes_to_print*>,<*matches_to_display*>

This command searches for <pattern> in the memory segment from <start_address> to <end_address> (both inclusive), and outputs the first <matches_to_display> matching addresses. It also displays the next <bytes_to_print> bytes for the last match.

By default, the **find** command searches for the supplied pattern at every byte within the range. If you know the pattern is aligned to a particular byte boundary, you can specify <*memory_boundary*> to search once every <*memory_boundary*> bytes, for example, 4 or 8 bytes.

findnext
Finds the next instance of the last string passed to **find**. This command is used in conjunction with **find** or **findptr** command to continue searching for the next matches. It repeats the previous **find** or **findptr** command (depending on which command is most recently issued) starting from the last match.

findptr
<*pattern*>,<*start_address*>,<*end_address*>,<*memory_boundary*>,<*bytes_to_print*>,<*matches_to_display*>
Searches memory for the given pointer. **findptr** searches for <*pattern*> as a pointer in the memory segment from <*start_address*> to <*end_address*> (both inclusive), and outputs the first <*matches_to_display*> matching addresses that start at the corresponding <*memory_boundary*>. It also displays the next <*bytes_to_print*> bytes for the last match.

x/ (examine)
Passes number of items to display and unit size ('b' for byte (8 bit), 'h' for half word (16 bit), 'w' for word (32 bit), 'g' for giant word (64 bit)) to sub-command (for example x/12bd). This is similar to the use of the **x/** command in gdb (including use of defaults).

x/J [<*0xaddr*>|<*class_name*>**]**

Displays information about a particular object or all objects of a class. If given class name, all static fields with their values are printed, followed by all objects of that class with their fields and values. If given an object address (in hex), static fields for that object's class are not printed; the other fields and values of that object are printed along with its address.

Note: This command ignores the number of items and unit size passed to it by the x/ command.

x/D <*0xaddr*>

Displays the integer at the specified address, adjusted for the endianness of the architecture this dump file is from.

Note: This command uses the number of items and unit size passed to it by the x/ command.

x/X <*0xaddr*>

Displays the hex value of the bytes at the specified address, adjusted for the endianness of the architecture this dump file is from.

Note: This command uses the number of items and unit size passed to it by the x/ command.

x/K <*0xaddr*>

Displays the value of each section (whose size is defined by the pointer size of this architecture) of memory, adjusted for the endianness of the architecture this dump file is from, starting at the specified address. It also displays a module with a module section and an offset from the start of that module section in memory if the pointer points to that module section. If no symbol is found, it displays a "*" and an offset from the current address if the pointer points to an address within 4KB (4096 bytes) of the current address. While this command can work on an arbitrary section of memory, it is probably most useful when used on a section of memory that refers to a stack frame. To find the memory section of a thread's stack frame, use the info thread command.

Note: This command uses the number of items and unit size passed to it by the **x/** command.

deadlock

Displays information about deadlocks if there are any set. This command shows detailed information about deadlocks or "no deadlocks detected" if there are no deadlocks. A deadlock situation consists of one or more deadlock loops and zero or more branches attached to those loops. This command prints out each branch attached to a loop and then the loop itself. If there is a split in a deadlock branch, separate branches are created for each side of the split in the branch. Deadlock branches start with a monitor that has no threads waiting on it and the continues until it reaches a monitor that exists in another deadlock branch or loop. Deadlock loops start and end with the same monitor.

Monitors are represented by their owner and the object associated with the given monitor. For example, the **3435 (0x45ae67)** command represents the monitor that is owned by the thread with id 3435 and is associated the object at address 0x45ae67. Objects can be viewed by using a command like **x/j 0x45ae67** and threads can be viewed using a command like **info thread 3435**.

set logging

Configures several logging-related parameters, starts/stops logging. This allows the results of commands to be logged to a file.

The values are: [on | off], file <filename>, overwrite [on | off], redirect [on | off]
- [on | off] - turns logging on or off (default: off)
- file <filename> - sets the file to log to; this will be relative to the directory returned by the pwd command unless an absolute path is specified; if the file is set while logging is on, the change will take effect the next time logging is started (default: <not set>)
- overwrite [on | off] - turns overwriting of the specified log file on or off, off means that data will be appended to the log file; if this is on, the log file is cleared every time the set logging on command is run (default: off)
- redirect [on | off] - turns redirecting to file on or off (on means that non-error output goes only to the log file (not the console) when logging is on, off means that non-error output goes to both the console and the log file); redirection must be turned off logging can be turned off (default: off)

show logging
Displays the current values of the logging settings command:
- set_logging = [on | off]
- set_logging_file =
- set_logging_overwrite = [on | off]
- set_logging_redirect = [on | off]
- current_logging_file = - file that is currently being logged to; it could be different than set_logging_file, if that value was changed after logging was started.

cd <*directory_name*>
Changes the current working directory, used for log files. Changes the current working directory to <*directory_name*>, checking to see if it exists and is a directory before making the change. The current working directory is where log files are outputted to; a change to the current working directory has no effect on the current log file setting because the logging filename is converted to an absolute path when set. Note: to see what the current working directory is set to, use the **pwd** command.

pwd
Displays the current working directory which is the directory where log files are stored.

quit
Exits the core file viewing tool; any log files that are currently open are closed before exit.

Example session

This example session illustrates a selection of the commands available and their use. The session shown is from a Linux dump. The output from other types of dump is substantially the same.

In the example session, some lines have been removed for clarity (and terseness). Some comments (contained within braces) are included to explain various aspects with some comments on individual lines looking like:

```
{ comment }
```

User input is prefaced by a ">".

```
{First, invoke DTFJview via the jdmpview launcher, using the -Xrealtime flag and passing in the name of a dump.}

> jdmpview -Xrealtime -core ~/deadlock/deadlock/core.20060731.171834.28825.dmp

Loading image from DTFJ...

DTFJ View version 1.0.6
```

```
Using DTFJ API version 1.1
For a list of commands, type "help"; for how to use "help", type "help help"


{the output produced by help is illustrated below}

>help

        info
         thread   displays information about Java and native threads
         system   displays information about the system the core dump is from
         class    prints inheritance chain and other data for a given class
         proc     displays threads, command line arguments, environment variables,
                                                            and shared modules of current process
         jitm     displays JIT'ed methods and their addresses
         ls       outputs a list of available monitors and locked objects
         sym      outputs a list of available modules
         mmap     outputs a list of all memory segments in the address space
         heap     displays information about Java heaps
        hexdump   outputs a section of memory in a hexdump-like format
        +         displays the next section of memory in hexdump-like format
        -         displays the previous section of memory in hexdump-like format
        find      searches memory for a given string
        deadlock  displays information about deadlocks if there are any
        set
         logging  configures several logging-related parameters, starts/stops logging
        show
         logging  displays the current values of logging settings
        quit      exits the core file viewing tool
        whatis    gives information about what is stored at the given memory address
        cd        changes the current working directory, used for log files
        pwd       displays the current working directory
        findnext  finds the next instance of the last string passed to "find"
        findptr   searches memory for the given pointer
        help      displays list of commands or help for a specific command
        x/        works like "x/" in gdb (including use of defaults): passes number of items to display
                       and unit size ('b' for byte, 'h' for halfword, 'w' for word, 'g' for giant word)
                       to sub-command (ie. x/12bd)
         j        displays information about a particular object or all objects of a class
         d        displays the integer at the specified address
         x        displays the hex value of the bytes at the specified address
         k        displays the specified memory section as if it were a stack frame



{In jdmpview setting an output file could be done from the invocation, in DTFJView it must be
                                                                  done using the "set logging" command. }

> set logging file log.txt

        log file set to "log.txt"

> set logging on

        logging turned on; outputting to "/home/test/log.txt"

> show logging

        set_logging = on
        set_logging_file = "log.txt"
        set_logging_overwrite = off
        set_logging_redirect = off

        current_logging_file = "/home/test/log.txt"



>info thread   << Displays info on current thread. Use "info thread *" for information on all threads.


        native threads for address space # 0
         process id: 28836

          thread id: 28836
           registers:
            cs       = 0x00000073   ds       = 0x0000007b   eax      = 0x00000000   ebp      = 0xbfe32064
            ebx      = 0xb7e9e484   ecx      = 0x00000000   edi      = 0xbfe3245c   edx      = 0x00000002
```

```
            efl      = 0x00010296   eip      = 0xb7e89120   es       = 0xc010007b   esi      = 0xbfe32471
            esp      = 0xbfe31c2c   fs       = 0x00000000   gs       = 0x00000033   ss       = 0x0000007b
           stack sections:
            0xbfe1f000 to 0xbfe34000 (length 0x15000)
           stack frames:
            bp: 0xbfe32064   proc name: /home/test/sdk/jre/bin/java::_fini

 ==== lines removed for terseness====
 ==== lines removed for terseness====

            bp: 0x00000000   proc name: <unknown location>
           properties:

           associated Java thread: <no associated Java thread>


>info system

        System Summary
        ==============

        System Memory:  2323206144 bytes
        System:         Linux
        Virtual Machine(s):
                # 0: Java(TM) 2 Runtime Environment, Standard Edition(build 2.3)
IBM J9 VM(J2RE 1.5.0 IBM J9 2.3 Linux x86-32 j9vmxi3223ifx-20060719 (JIT enabled)
J9VM - 20060714_07194_lHdSMR
JIT  - 20060428_1800.ifix2_r8
GC   - 200607_07)


> info class  << Information on all classes

 runtime #1 - version: Java(TM) 2 Runtime Environment, Standard Edition(build 2.3)
IBM J9 VM(J2RE 1.5.0 IBM J9 2.3 Linux x86-32 j9vmxi3223ifx-20060719 (JIT enabled)
J9VM - 20060714_07194_lHdSMR
JIT  - 20060428_1800.ifix2_r8
GC   - 200607_07)
        instances      total size  class name
                0               0  java/util/regex/Pattern$Slice
                0               0  java/lang/Byte
                0               0  java/lang/CharacterDataLatin1
                2              96  sun/nio/cs/StreamEncoder$ConverterSE
             1015           36540  java/util/TreeMap$Entry

 ==== lines removed for terseness====
 ==== lines removed for terseness====

                 2              48  [java/io/File
                5             104  [java/io/ObjectStreamField
                0               0  java/lang/StackTraceElement

  Total number of objects: 9240
  Total size of objects: 562618



> info class java/util/Random  << Information on a specific class

        runtime #1 - version: Java(TM) 2 Runtime Environment, Standard Edition(build 2.3)
IBM J9 VM(J2RE 1.5.0 IBM J9 2.3 Linux x86-32 j9vmxi3223ifx-20060719 (JIT enabled)
J9VM - 20060714_07194_lHdSMR
JIT  - 20060428_1800.ifix2_r8
GC   - 200607_07)
name = java/util/Random

        ID = 0x81c9fb0    superID = 0x80ea450
        classLoader = 0x82307e8    modifers: public synchronized

        number of instances:     1
        total size of instances: 32 bytes

Inheritance chain....

        java/lang/Object
           java/util/Random

Fields......
```

```
          static fields for "java/util/Random"
           static final long serialVersionUID = 3905348978240129619 (0x363296344bf00a53)
           private static final long multiplier = 25214903917 (0x5deece66d)
           private static final long addend = 11 (0xb)

 ==== lines removed for terseness====

          non-static fields for "java/util/Random"
           private long seed
           private double nextNextGaussian
           private boolean haveNextNextGaussian


Methods......

Bytecode range(s): 81fb41c -- 81fb430:  public void <init>()

 ==== lines removed for terseness====

Bytecode range(s): 81fb624 -- 81fb688:  public synchronized double nextGaussian()
Bytecode range(s): 81fb69c -- 81fb6a4:  static void <clinit>()


> info proc

        address space # 0

         Thread information for current process:
          Thread id: 28836

         Command line arguments used for current process:
          /home/test/sdk/jre/bin/java -Xmx100m -Xms100m -Xtrace: DeadlockCreator

         Environment variables for current process:
          IBM_JAVA_COMMAND_LINE=/home/test/sdk/jre/bin/java -Xmx100m -Xms100m -Xtrace: DeadlockCreator
          LIBPATH=.:/home/test/sdk/jre/bin:/home/test/sdk/jre/bin/j9vm:/usr/bin/gcc:
          HISTSIZE=1000

 ==== lines removed for terseness====

           PATH=.:/home/test/sdk/bin:/home/test/sdk/jre/bin/j9vm:/home/test/sdk/jre/bin:
                /usr/bin/gcc:/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/test/bin
          TERM=xterm



> info jitm

        start=0xb11e241c  end=0xb11e288f   DeadlockCreator::main([Ljava/lang/String;)V
        start=0xb11e28bc  end=0xb11e64ca   DeadlockThreadA::syncMethod(LDeadlockThreadA;)V
        start=0xb11e64fc  end=0xb11ea0fa   DeadlockThreadB::syncMethod(LDeadlockThreadB;)V
        start=0xb11dd55c  end=0xb11dd570   java/lang/Object::<init>()V

   ==== lines removed for terseness====
 ==== lines removed for terseness====

        start=0xb11e0f54  end=0xb11e103e   java/util/zip/ZipEntry::initFields(J)V
        start=0xb11e0854  end=0xb11e0956   java/util/zip/Inflater::inflateBytes([BII)I
        start=0xb11e13d4  end=0xb11e14bf   java/util/zip/Inflater::reset(J)V

> info ls
(un-named monitor @0x835ae50 for object @0x835ae50)
 owner thread's id = <data unavailable>
 object = 0x835ae50

(un-named monitor @0x835b138 for object @0x835b138)
 owner thread's id = <data unavailable>
 object = 0x835b138

Thread global
Raw monitor: id = <unavailable>

NLS hash table
Raw monitor: id = <unavailable>

portLibrary_j9sig_sync_monitor
Raw monitor: id = <unavailable>
```

```
portLibrary_j9sig_asynch_reporter_shutdown_monitor
Raw monitor: id = <unavailable>


 ==== lines removed for terseness====
 ==== lines removed for terseness====

Thread public flags mutex
Raw monitor: id = <unavailable>

Thread public flags mutex
Raw monitor: id = <unavailable>

JIT-QueueSlotMonitor-21
Raw monitor: id = <unavailable>

Locked objects...
 java/lang/Class@0x835ae50 is locked by a thread with id <data unavailable>
 java/lang/Class@0x835b138 is locked by a thread with id <data unavailable>


> info sym

        modules for address space # 0
         process id: 28836



> info mmap
Address: 0x1000   size: 0x1000 (4096)
Address: 0x8048000   size: 0xd000 (53248)
Address: 0x8055000   size: 0x2000 (8192)
Address: 0x8057000   size: 0x411000 (4263936)

 ==== lines removed for terseness====
 ==== lines removed for terseness====

Address: 0xffffe460   size: 0x18 (24)
Address: 0xffffe478   size: 0x24 (36)
Address: 0xffffe5a8   size: 0x78 (120)


> info heap object <<Displays information on the object heap, "info heap *" displays
                                                                    information on all heaps.

        Runtime #1
         Heap #1:  Default
          Size of heap: 104857600 bytes
          Occupancy :   562618 bytes  (0.53%)
          Section #1:  Contiguous heap extent at 0xb1439000 (0x6400000 bytes)
           Size:        104857600 bytes
           Shared:      false
           Executable:  false
           Read Only:   false

> hexdump 0xb1439000 200

b1439000: 70da0e08 0e800064 00000000 00000000  |p......d........|
b1439010: d0c20e08 05800464 00000000 80000000  |.......d........|
b1439020: 00000100 02000300 04000500 06000700  |................|
b1439030: 08000900 0a000b00 0c000d00 0e000f00  |................|
b1439040: 10001100 12001300 14001500 16001700  |................|
b1439050: 18001900 1a001b00 1c001d00 1e001f00  |................|
b1439060: 20002100 22002300 24002500 26002700  | .!."#.$.%.&.'.|
b1439070: 28002900 2a002b00 2c002d00 2e002f00  |(.).*.+.'.-.../.|
b1439080: 30003100 32003300 34003500 36003700  |0.1.2.3.4.5.6.7.|
b1439090: 38003900 3a003b00 3c003d00 3e003f00  |8.9.:.;.<.=.>.?.|
b14390a0: 40004100 42004300 44004500 46004700  |@.A.B.C.D.E.F.G.|
b14390b0: 48004900 4a004b00 4c004d00 4e004f00  |H.I.J.K.L.M.N.O.|
b14390c0: 50005100 52005300                    |P.Q.R.S.

> +

b14390c8: 54005500 56005700 58005900 5a005b00  |T.U.V.W.X.Y.Z.[.|
b14390d8: 5c005d00 5e005f00 60006100 62006300  |\.].^._.`.a.b.c.|
b14390e8: 64006500 66006700 68006900 6a006b00  |d.e.f.g.h.i.j.k.|
b14390f8: 6c006d00 6e006f00 70007100 72007300  |l.m.n.o.p.q.r.s.|
b1439108: 74007500 76007700 78007900 7a007b00  |t.u.v.w.x.y.z.{.|
b1439118: 7c007d00 7e007f00 e0df0e08 01804864  ||.}.~.........Hd|
```

```
b1439128: 00000000 00000000 90e00e08 01804c64  |..............Ld|
b1439138: 00000000 00000000 78f30e08 0e805064  |........x.....Pd|
b1439148: 00000000 b89b43b1 b89b43b1 00000000  |......C...C.....|
b1439158: 78f30e08 0e805664 00000000 109c43b1  |x.....Vd......C.|
b1439168: 109c43b1 00000000 78f30e08 0e805c64  |..C.....x.....\d|
b1439178: 00000000 709c43b1 709c43b1 00000000  |....p.C.p.C.....|
b1439188: 78f30e08 0e806264                    |x.....bd
```

> -

```
b1439000: 70da0e08 0e800064 00000000 00000000  |p......d........|
b1439010: d0c20e08 05800464 00000000 80000000  |.......d........|
b1439020: 00000100 02000300 04000500 06000700  |................|
b1439030: 08000900 0a000b00 0c000d00 0e000f00  |................|
b1439040: 10001100 12001300 14001500 16001700  |................|
b1439050: 18001900 1a001b00 1c001d00 1e001f00  |................|
b1439060: 20002100 22002300 24002500 26002700  | .!."#.$.%.&.'.|
b1439070: 28002900 2a002b00 2c002d00 2e002f00  |(.).*.+.'.-.../.|
b1439080: 30003100 32003300 34003500 36003700  |0.1.2.3.4.5.6.7.|
b1439090: 38003900 3a003b00 3c003d00 3e003f00  |8.9.:.;.<.=.>.?.|
b14390a0: 40004100 42004300 44004500 46004700  |@.A.B.C.D.E.F.G.|
b14390b0: 48004900 4a004b00 4c004d00 4e004f00  |H.I.J.K.L.M.N.O.|
b14390c0: 50005100 52005300                    |P.Q.R.S.
```

> whatis 0xb143a000

```
        runtime #1 - version: Java(TM) 2 Runtime Environment, Standard Edition(build 2.3)
IBM J9 VM(J2RE 1.5.0 IBM J9 2.3 Linux x86-32 j9vmxi3223ifx-20060719 (JIT enabled)
J9VM - 20060714_07194_lHdSMR
JIT  - 20060428_1800.ifix2_r8
GC   - 200607_07)
         heap #1 - name: object heap

0xb143a000 is within the heap segment: b1439000 -- b7839000
0xb143a000 is within an object on the heap.
        Offset 8 within [char instance @ 0xb1439ff8
```

```
{ find command parameters are: <pattern>,<start_address>,<end_address>,<memory_boundary>,
                                                    <bytes_to_print>,<matches_to_display> }

> find a,0b1439000,0xb1440000,10,20,5
#0: 0xb1439c00
#1: 0xb1439c46
#2: 0xb143a1be
#3: 0xb143a1c8
#4: 0xb143a1e6

b143a1e6: 61007000 70006500 6e006900 6e006700  |a.p.p.e.n.i.n.g.|
b143a1f6: 45007800                             |E.x.

> findnext <<Repeats find command, starting from last match.
#0: 0xb143a72c
#1: 0xb143b3f2
#2: 0xb143b47e
#3: 0xb143b492
#4: 0xb143b51e

b143b51e: 61002e00 73007000 65006300 69006600  |a...s.p.e.c.i.f.|
b143b52e: 69006300                             |i.c.

> findnext
#0: 0xb143b532
#1: 0xb143b5e6
#2: 0xb143b5fa
#3: 0xb143b71c
#4: 0xb143bac8

b143bac8: 61007000 00000000 10cd0e08 0e80b46e  |a.p............n|
b143bad8: 00000000                             |....
```

```
{ x/j can be passed an object address or a class name }
> x/j 0xb1439000

        runtime #1 - version: Java(TM) 2 Runtime Environment, Standard Edition(build 2.3)
IBM J9 VM(J2RE 1.5.0 IBM J9 2.3 Linux x86-32 j9vmxi3223ifx-20060719 (JIT enabled)
J9VM - 20060714_07194_lHdSMR
```

```
JIT  - 20060428_1800.ifix2_r8
GC   - 200607_07)
         heap #1 - name: object heap

          java/lang/String$CaseInsensitiveComparator @ 0xb1439000

{If passed an object address the (non-static) fields and values of the object will be printed }

> x/j java/lang/Float

        runtime #1 - version: Java(TM) 2 Runtime Environment, Standard Edition(build 2.3)
IBM J9 VM(J2RE 1.5.0 IBM J9 2.3 Linux x86-32 j9vmxi3223ifx-20060719 (JIT enabled)
J9VM - 20060714_07194_lHdSMR
JIT  - 20060428_1800.ifix2_r8
GC   - 200607_07)
         heap #1 - name: object heap

          static fields for "java/lang/Float"
           public static final float POSITIVE_INFINITY = Infinity (0x7f800000)
           public static final float NEGATIVE_INFINITY = -Infinity (0xffffffffff800000)
           public static final float NaN = NaN (0x7fc00000)
           public static final float MAX_VALUE = 3.4028235E38 (0x7f7fffff)
           public static final float MIN_VALUE = 1.4E-45 (0x1)
           public static final int SIZE = 32 (0x20)
           public static final Class TYPE = <object> @ 0x80ec368
           private static final long serialVersionUID = -2671257302660747028 (0xdaedc9a2db3cf0ec)

          <no object of class "java/lang/Float" exists>

{If passed a class name the static fields and their values are printed, followed by all objects of
                                                                                          that class }


> x/d 0xb1439000

        0xb1439000: 135191152 <<Integer at specified address


> x/x 0xb1439000

        0xb1439000: 080eda70 <<Hex value ofthe bytes at specified address


{ "cd" and "pwd" are self explanatory. }
> pwd
        /home/test

> cd deadlock/
> pwd

        /home/test/deadlock

> quit
```

jdmpview -Xrealtime commands quick reference

A short list of the commands you use with **jdmpview -Xrealtime**.

Table 22. jdmpview -WebSphere Real Time commands - quick reference

Command	Sub-command	Description
help		Displays a list of commands or help for a specific command.
info		
	thread	Displays information about Java and native threads.
	system	Displays information about the system the core dump is from.
	class	Displays the inheritance chain and other data for a given class.
	proc	Displays threads, command line arguments, environment variables, and shared modules of current process.

Table 22. jdmpview -WebSphere Real Time commands - quick reference (continued)

Command	Sub-command	Description
	jitm	Displays JITed methods and their addresses.
	ls	Displays a list of available monitors and locked objects.
	sym	Displays a list of available modules.
	mmap	Displays a list of all memory segments in the address space.
	heap	Displays information about all heaps or the specified heap.
hexdump		Displays a section of memory in a hexdump-like format.
+		Displays the next section of memory in hexdump-like format.
-		Displays the previous section of memory in hexdump-like format.
whatis		Displays information about what is stored at the given memory address.
find		Searches memory for a given string.
findnext		Finds the next instance of the last string passed to “find”.
findptr		Searches memory for the given pointer.
x/ (examine)		Examine works like ″x/″ in gdb (including use of defaults): passes number of items to display and unit size (’b’ for byte (8 bit), ’h’ for half word (16 bit), ’w’ for word (32 bit), ’g’ for giant word (64 bit)) to sub-command (for example x/12bd).
	J	Displays information about a particular object or all objects of a class.
	D	Displays the integer at the specified address.
	X	Displays the hex value of the bytes at the specified address.
	K	Displays the specified memory section as if it were a stack frame.
deadlock		Displays information about deadlocks if there are any set.
set logging		Configures several logging-related parameters, starts/stops logging. This allows the results of commands to be logged to a file.
show logging		Displays the current values of logging settings.
cd		Changes the current working directory, used for log files.
pwd		Displays the current working directory.
quit		Exits exits the core file viewing tool; any log files that are currently open will be closed prior to exit.

Problem determination for compilers

The Just-In-Time (JIT) and Ahead-of-Time (AOT) compilers are tightly bound to the JVM, but are not part of it. The compilers convert Java bytecodes, which are interpreted by the JVM at run time and are executed slowly, into native code. Native code is understood by the processor and executes quickly.

Occasionally, valid bytecodes might compile into invalid native code, causing the Java program to fail. By determining whether the compiler is faulty and, if so, where it is faulty, you can provide valuable help to the Java service team.

To determine what methods are compiled, and at what optimization level at AOT build time, use the **-Xjit:verbose** option, for example, `jxeinajar -Xjit:verbose -outPath myAOTDir my.jar`

This section describes how you can determine if your problem is JIT-related. This section also suggests some possible workarounds and debugging techniques for solving JIT-related problems:

Disabling the JIT

The JIT is enabled by default, but for efficiency reasons, not all methods in a Java application are compiled. The JVM maintains a call count for each method in the application; every time a method is called and interpreted, the call count for that method is incremented. When the count reaches the JIT threshold, the method is compiled and executed natively.

The call count mechanism spreads compilation of methods throughout the life of an application, giving higher priority to methods that are used most frequently. Some infrequently used methods might never be compiled at all. As a result, when a Java program fails, the problem might be in the JIT, or it might be elsewhere in the JVM. The first step in diagnosing the failure is to determine *where* the problem is. To do this, you must first run your Java program in purely interpreted mode (that is, with the JIT disabled): specify the **-Xint** option, and remove the **-Xjit** option (and accompanying JIT parameters, if any) when you run the JVM. If the failure still occurs, the problem is most likely in the JVM rather than the JIT. (Do not use the **-Xint** and the **-Xjit** options together.)

Running the Java program with the JIT disabled leads to one of the following:

- The failure remains. The problem is, therefore, not in the JIT. Do not read further in this section. In some cases, the program might start failing in a different manner; nevertheless, the problem is not related to the JIT.
- The failure disappears. The problem is most likely, although not definitely, in the JIT.

Selectively disabling the JIT

If the failure of your Java program appears to come from a problem within the JIT, you can try to narrow down the problem further.

The JIT optimizes methods at various optimization levels; that is, different selections of optimizations are applied to different methods, based on their call counts. Methods that are called more frequently are optimized at higher levels. By changing JIT parameters, you can control the optimization level at which methods are optimized, and determine whether the optimizer is at fault and, if it is, which optimization is problematic.

JIT parameters are specified as a comma-separated list, appended to the **-Xjit** option. The syntax is `-Xjit:param1,param2=value,...`. For example,

```
java -Xjit:verbose,optLevel=noOpt HelloWorld
```

runs the HelloWorld program, while enabling verbose output from the JIT, and making it generate native code without performing any of the optimizations listed in Diagnostics Guide.

The JIT parameters give you a powerful tool that enables you to determine the location of a JIT problem; whether it is in the JIT itself or in some Java code that causes the JIT to fail. In addition, when you have identified a problem area, you are automatically given a workaround so that you can continue to develop or deploy code while losing only a fraction of JVM performance.

The first JIT parameter to try is **count=0**, which sets the JIT threshold to zero and effectively causes the Java program to be run in purely compiled mode.

If the failure still occurs, add **disableInlining** to the command line. With this parameter set, the JIT is prohibited from generating larger and more complex code in an attempt to perform aggressive optimizations.

If the failure persists, try decreasing JIT optimization levels. The various optimization levels are:

1. scorching
2. veryHot
3. hot
4. warm
5. cold
6. noOpt

Run the Java program with:

```
-Xjit:count=0,disableInlining,optLevel=scorching
```

Try each of the optimization levels in turn, and record your observations. If one of these settings causes your failure to disappear, you have a quick workaround that you can use while the Java service team analyzes and fixes the JIT problem. If you can remove **disableInlining** from the JIT parameter list (that is, if removing it does not cause the failure to reappear), do so to improve performance.

Locating the failing method

When you have arrived at the lowest optimization level at which the JIT must compile methods to trigger the failure, you can try to find out which part of the Java program, when compiled, causes the failure.

You can then instruct the JIT to limit the workaround to a specific method, class, or package, allowing the JIT to compile the rest of the program as it normally would.

If the failure occurs with **optLevel=noOpt**, you can also instruct the JIT to not compile the method or methods that are causing the failure (thus avoiding it).

To locate the method that is causing the failure, follow these steps:

1. Run the Java program with the JIT parameters **verbose** and **vlog=***filename*. With these parameters, the JIT reports its progress, as it compiles methods, in a verbose log file, also called a *limit file*. A typical limit file contains lines that correspond to compiled methods, like:

   ```
   + (hot) java/lang/Math.max(II)I @ 0x10C11DA4-0x10C11DDD
   ```

   Lines that do not start with the plus sign are ignored by the JIT in the steps below, so you can edit them out of the file.

2. Run the program again with the JIT parameter **limitFile=(filename,m,n)**, where **filename** is the path to the limit file, and **m** and **n** are line numbers indicating respectively the first and the last methods in the limit file that should be compiled. This parameter causes the JIT to compile only the methods listed on lines m to n in the limit file. Methods not listed in the limit file and methods listed on lines outside the range are not compiled. Repeat this step with a smaller range if the program still fails.

   The recommended number of lines to select from the limit file in each repetition is half of the previous selection, so that this step is essentially a binary search for the failing method.

3. If the program no longer fails, one or more of the methods that you have removed in the last iteration must have been the cause of the failure. Select methods not selected in the previous iteration and repeat the previous step to see if the program fails again.

4. Repeat the last two steps, as many times as necessary, to find the minimum number of methods that must be compiled to trigger the failure. Often, you can reduce the file to a single line.

When you have obtained a workaround and located the failing method, you can limit the workaround to the failing method. For example, if the method java/lang/Math.max(II)I causes the program to fail when compiled with **optLevel=hot**, you can run the program with:

```
-Xjit:{java/lang/Math.max(II)I}(optLevel=warm,count=0)
```

which tells the JIT to compile only the troublesome method at an optimization level of "warm", but compile all other methods normally.

If a method fails when it is compiled at "noOpt", you can exclude it from compilation altogether, using the **exclude=**<*method*> parameter:

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

Identifying JIT compilation failures

If the JVM crashes, and you can see that the failure has occurred in the JIT library (libj9jit23.so), the JIT might have failed during an attempt to compile a method.

To see if the JIT is crashing in the middle of a compilation, use the **verbose** option with the following additional settings:

```
-Xjit:verbose='{compileStart|compileEnd}'
```

These **verbose** settings report when the JIT starts to compile a method, and when it ends. If the JIT fails on a particular method (that is, it starts compiling, but crashes before it can end), use the **exclude=** parameter to exclude it from compilation (refer to "Locating the failing method" on page 155). If excluding the method prevents the crash, you have an excellent workaround that you can use while the service team corrects your problem.

Identifying AOT compilation failures

AOT problem determination is very similar to JIT problem determination.

As with the JIT, a good first test is to run with **-Xnojit** which ensures that the AOT'ed code is not used when running the application. If this makes the program function normally, you can rebuild the AOT'ed jar files using the same technique as described in Locating the failing method, except that you will provide the **-Xjit** option at AOT build time instead of application runtime.

Performance of short-running applications

The IBM JIT is tuned for long-running applications typically used on a server.

If the performance of short-running applications is worse than expected, try the **-Xquickstart** command-line parameter (refer to the **-Xquickstart** option in Diagnostics Guide), especially for those applications in which execution time is not concentrated into a small number of methods.

Also try adjusting the JIT threshold (using trial and error) for short-running applications to improve performance. Refer to "Selectively disabling the JIT" on page 154.

Garbage Collector diagnostics

This Garbage Collection policy is only used by Standard Java.

Note: If you are using **-Xrealtime**, see "Troubleshooting the Metronome Garbage Collector" on page 23 otherwise refer to the Diagnostics Guide.

When a garbage collection cycle starts, the Garbage Collector locates all objects in the heap that are still in use or "live". When this has been done, any objects that are not in the list of "live" objects are unreachable. They are garbage, and can be collected.

The key here is the condition *unreachable*. The Garbage Collector traces all references that an object makes to other objects. Any such reference automatically means that an object is *reachable* and not garbage. So, if the objects of an application make reference to other objects, those other objects are live and cannot be collected. However, obscure references sometimes exist that the application overlooks. These references are reported as memory leaks.

Listeners

By installing a listener, you effectively attach your object to a static reference that is in the listener. Your object cannot be collected while the listener is active. You must explicitly uninstall a listener when you have finished using the object to which you attached it.

Hash Tables

Anything that is added to a hash table, either directly or indirectly, from an instance of your object, creates a reference to your object from the hashed object. Hashed objects cannot be collected unless they are explicitly removed from any hash table to which they have been added.

Hash tables are common causes of perceived leaks. If an object is placed into a hash table, that object and all the objects that it references are reachable.

Static Data

This exists independently of instances of your object. Anything that it points to cannot be collected even if no instances of your class are present that contain the static data.

JNI references

Objects that are passed from the JVM to an application across the JNI interface have a reference to them that is held in the JNI code of the JVM. Without this reference, the Garbage Collector cannot trace live native objects. Such references must be explicitly cleared by the native code application before they can be collected. See the JNI documentation on the Sun website (java.sun.com) for more information.

Premature expectation

You instantiate a class, finish with it, tidy up all listeners, and so on. You have a finalizer in the class, and you use that finalizer to report that the finalizer has been

called. On all the later garbage collection cycles, your finalizer is not called. It seems that your unused object is not being collected and that a memory leak has occurred, but this is not so.

The Garbage Collector does *not* collect garbage unless it needs to. It does not necessarily collect all garbage when it does run. It might not collect garbage if you manually invoke it (by using System.gc()). This is because running the Garbage Collector is an intensive operation, and it is designed to run as infrequently as possible for as short a time as possible.

Objects with finalizers

Objects that have finalizers cannot be collected until the finalizer has run. Finalizers run on a separate thread, and thus their execution might be delayed, or not occur at all. This is allowed. See Diagnostics Guide for more details.

Using DTFJ

The Diagnostic Tooling Framework for Java (DTFJ) is a Java application programming interface (API) from IBM used to support the building of Java diagnostics tools.

You process the dumps passed to DTFJ with the jextract tool; see jextract. The jextract tool produces metadata from the dump, which allows the internal structure of the JVM to be analyzed. jextract must be run on the system that produced the dump.

The DTFJ API helps diagnostics tools access the following information:

- Memory locations stored in the dump
- Relationships between memory locations and Java internals
- Java threads running within the JVM
- Native threads held in the dump
- Java classes and objects that were present in the heap
- Details of the machine on which the dump was produced
- Details of the Java version that was being used
- The command line that launched the JVM

DTFJ is implemented in pure Java and tools written using DTFJ can be cross-platform. Therefore, it is possible to analyze a dump taken from one machine on another (remote and more convenient) machine. For example, a dump produced on an AIX PPC machine can be analyzed on a Windows Thinkpad.

The full details of the DTFJ Interface are provided with the SDK as Javadoc in sdk/docs/apidoc.zip. DTFJ classes are accessible without modification to the class path.

This section describes DTFJ in:

- "Overview of the DTFJ interface"
- "DTFJ example application" on page 163

Overview of the DTFJ interface

To create applications that use DTFJ, you must use the DTFJ interface.

Figure 14 on page 162 illustrates the DTFJ interface. The starting point for working with a dump is to obtain an Image instance by using the ImageFactory class supplied with the concrete implementation of the API.

The following example shows how to work with a dump.

```
import java.io.File;
import java.util.Iterator;
import java.io.IOException;

import com.ibm.dtfj.image.CorruptData;
import com.ibm.dtfj.image.Image;
import com.ibm.dtfj.image.ImageFactory;

public class DTFJEX1 {
    public static void main(String[] args) {
        Image image = null;
        if (args.length > 0) {
            File f = new File(args[0]);
```

```
                try {
                    Class factoryClass = Class
                            .forName("com.ibm.dtfj.image.j9.ImageFactory");
                    ImageFactory factory = (ImageFactory) factoryClass
                            .newInstance();
                    image = factory.getImage(f);
                } catch (ClassNotFoundException e) {
                    System.err.println("Could not find DTFJ factory class");
                    e.printStackTrace(System.err);
                } catch (IllegalAccessException e) {
                    System.err.println("IllegalAccessException for DTFJ factory class");
                    e.printStackTrace(System.err);
                } catch (InstantiationException e) {
                    System.err.println("Could not instantiate DTFJ factory class");
                    e.printStackTrace(System.err);
                } catch (IOException e) {
                    System.err.println("Could not find/use required file(s)");
                    e.printStackTrace(System.err);
                }
            } else {
                System.err.println("No filename specified");
            }
            if (image == null) {
                return;
            }

            Iterator asIt = image.getAddressSpaces();
            int count = 0;
            while (asIt.hasNext()) {
                Object tempObj = asIt.next();
                if (tempObj instanceof CorruptData) {
                    System.err.println("Address Space object is corrupt: "
                            + (CorruptData) tempObj);
                } else {
                    count++;
                }
            }
            System.out.println("The number of address spaces is: " + count);
        }
    }
```

In this example, the only section of code that ties the dump to a particular implementation of DTFJ is the generation of the factory class - it would be a straightforward task to amend this code to cope with handling different implementations.

The getImage() methods in ImageFactory expect one file, the dumpfilename.zip file produced by JExtract. If the getImage() methods are called with two files, they are interpreted as the dump itself and the .xml metadata file. If there is a problem with the file specified, an IOException is thrown by getImage() and can be caught and (in the example above) an appropriate message issued. If a missing file was passed to the above example, the following output would be produced:

```
Could not find/use required file(s)
java.io.FileNotFoundException: core_file.xml (The system cannot find the file specified.)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:135)
        at com.ibm.dtfj.image.j9.ImageFactory.getImage(ImageFactory.java:47)
        at com.ibm.dtfj.image.j9.ImageFactory.getImage(ImageFactory.java:35)
        at DTFJEX1.main(DTFJEX1.java:23)
```

In the case above, the DTFJ implementation is expecting a dump file to exist. Different errors would be caught if the file existed but was not recognized as a valid dump file.

After you have obtained an Image instance, you can begin analyzing the dump. The Image instance is the second instance in the class hierarchy for DTFJ illustrated by Figure 14.

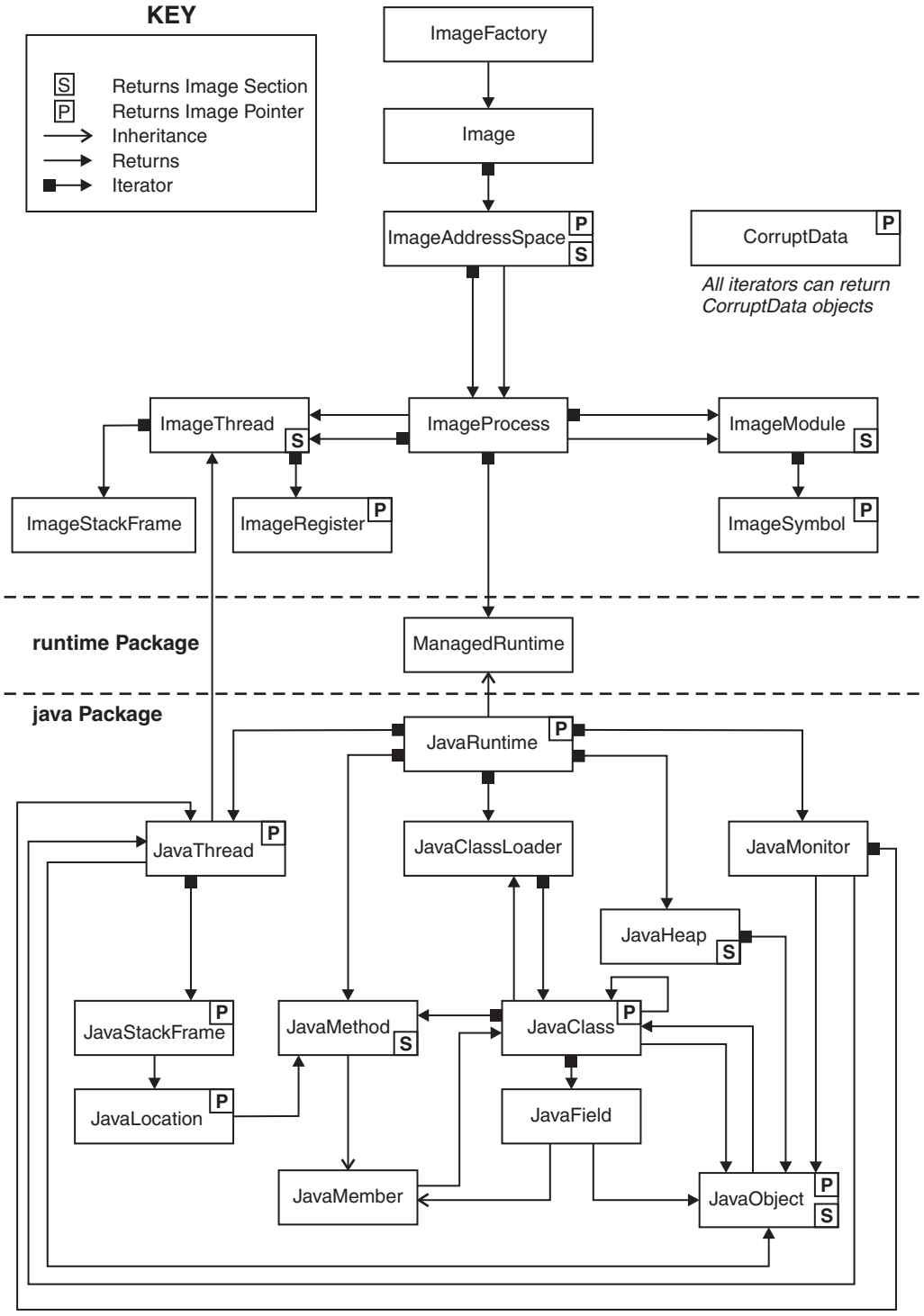

Figure 14. Diagram of the DTFJ interface

The hierarchy displays some major points of DTFJ. Firstly, there is a separation between the Image (the dump, a sequence of bytes with different contents on different platforms) and the Java internal knowledge.

Some things to note from the diagram:

- The DTFJ interface is separated into two parts: classes with names that start with *Image* and classes with names that start with *Java*.
- *Image* and *Java* classes are linked using a ManagedRuntime (which is extended by JavaRuntime).
- An Image object contains one ImageAddressSpace object.
- An ImageAddressSpace object contains one ImageProcess object.
- Conceptually, you can apply the Image model to any program running with the ImageProcess, although for the purposes of this document discussion is limited to the IBM JVM implementations.

DTFJ example application

This example is a fully working DTFJ application.

For clarity, this example does not perform full error checking when constructing the main Image object and does not perform CorruptData handling in all of the iterators. In a production environment, you would use the techniques illustrated in the example in the "Overview of the DTFJ interface" on page 160.

In this example, the program iterates through every available Java thread and checks whether it is equal to any of the available image threads. When they are found to be equal, the program declares that it has, in this case, `"Found a match"`.

The example demonstrates:

- How to iterate down through the class hierarchy.
- How to handle CorruptData objects from the iterators.
- The use of the equals method for testing equality between objects.

```
import java.io.File;
import java.util.Iterator;
import com.ibm.dtfj.image.CorruptData;
import com.ibm.dtfj.image.CorruptDataException;
import com.ibm.dtfj.image.DataUnavailable;
import com.ibm.dtfj.image.Image;
import com.ibm.dtfj.image.ImageAddressSpace;
import com.ibm.dtfj.image.ImageFactory;
import com.ibm.dtfj.image.ImageProcess;
import com.ibm.dtfj.java.JavaRuntime;
import com.ibm.dtfj.java.JavaThread;
import com.ibm.dtfj.image.ImageThread;

public class DTFJEX2
{
   public static void main( String[] args )
   {
      Image image = null;
      if ( args.length > 0 )
      {
         File f = new File( args[0] );
         try
         {
            Class factoryClass = Class
                  .forName( "com.ibm.dtfj.image.j9.ImageFactory" );
            ImageFactory factory = (ImageFactory) factoryClass.newInstance( );
            image = factory.getImage( f );
```

```
      }
      catch ( Exception ex )
      { /*
          * Should use the error handling as shown in DTFJEX1.
          */
         System.err.println( "Error in DTFJEX2" );
         ex.printStackTrace( System.err );
      }
   }
   else
   {
      System.err.println( "No filename specified" );
   }

   if ( null == image )
   {
      return;
   }

   MatchingThreads( image );
}

public static void MatchingThreads( Image image )
{
   ImageThread imgThread = null;

   Iterator asIt = image.getAddressSpaces( );
   while ( asIt.hasNext( ) )
   {
      System.out.println( "Found ImageAddressSpace..." );

      ImageAddressSpace as = (ImageAddressSpace) asIt.next( );

      Iterator prIt = as.getProcesses( );

      while ( prIt.hasNext( ) )
      {
         System.out.println( "Found ImageProcess..." );

         ImageProcess process = (ImageProcess) prIt.next( );

         Iterator runTimesIt = process.getRuntimes( );
         while ( runTimesIt.hasNext( ) )
         {
            System.out.println( "Found Runtime..." );
            JavaRuntime javaRT = (JavaRuntime) runTimesIt.next( );

            Iterator javaThreadIt = javaRT.getThreads( );

            while ( javaThreadIt.hasNext( ) )
            {
               Object tempObj = javaThreadIt.next( );
               /*
                * Should use CorruptData handling for all iterators
                */
               if ( tempObj instanceof CorruptData )
               {
                  System.out.println( "We have some corrupt data" );
               }
               else
               {
                  JavaThread javaThread = (JavaThread) tempObj;
                  System.out.println( "Found JavaThread..." );
                  try
                  {
                     imgThread = (ImageThread) javaThread.getImageThread( );
```

```
                        // Now we have a Java thread we can iterator
                        // through the image threads
                        Iterator imgThreadIt = process.getThreads( );

                        while ( imgThreadIt.hasNext( ) )
                        {
                           ImageThread imgThread2 = (ImageThread) imgThreadIt
                                 .next( );
                           if ( imgThread.equals( imgThread2 ) )
                           {
                              System.out.println( "Found a match:" );
                              System.out.println( "\tjavaThread "
                                    + javaThread.getName( )
                                    + " is the same as " + imgThread2.getID( ) );
                           }
                        }
                     }
                     catch ( CorruptDataException e )
                     {
                        System.err.println( "ImageThread was corrupt: "
                              + e.getMessage( ) );
                     }
                     catch ( DataUnavailable e )
                     {
                        System.out.println( "DataUnavailable: "
                              + e.getMessage( ) );
                     }
                  }
               }
            }
         }
      }
   }
}
```

Many DTFJ applications will follow similar models.

Chapter 12. Reference

This set of topics lists the options and class libraries that can be used with WebSphere Real Time

Options

The launcher has a set of standard options that are supported on the current runtime environment and will be supported in future releases. In addition, there is a set of nonstandard options. The default options have been chosen for best general use.

Specifying Java options and system properties

There are three ways to specify Java properties and system properties.

You can specify Java options and system properties in these ways. In order of precedence, they are:

1. By specifying the option or property on the command line. For example:
   ```
   java -Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump MyJavaClass
   ```
2. By creating a file that contains the options, and specifying it on the command line using the **-Xoptionsfile=**<*filename*> option.
3. By creating an environment variable called **IBM_JAVA_OPTIONS** containing the options. For example:
   ```
   export IBM_JAVA_OPTIONS="-Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump"
   ```

Rightmost options on the command line have precedence over leftmost options; for example, if you specify the options **-Xint -Xjit myClass**, **-Xjit** takes precedence.

Real-time options

The definitions of **-Xrealtime** and **-Xbootclasspath/p:** options used in WebSphere Real Time.

The **-X** options listed below are for the WebSphere Real Time environment.

-Xrealtime

Starts the real-time mode. It is required if you want to run the Metronome Garbage Collector and use the Real-Time Specification for Java (RTSJ) services. If you do not specify this option, you will use the standard IBM SDK and Runtime Environment for Linux Platforms, Java 2 Technology, version 5.0.

-Xbootclasspath/p:<directories and zip/jar files separated by a colon>

Prepends the specified locations to the bootclass path. When precompiling jar files shipped by IBM, to ensure that the precompiled versions of the jar files are loaded instead of the originals, the precompiled jar files must be prepended to the bootclasspath.

There are also **-Xbootclasspath:** and **-Xbootclasspath/a:** options, to the Java runtime, for setting and appending to the bootclasspath respectively, but they are less commonly used and do not ensure that a precompiled jar file will be used.

Example

```
java -Xrealtime -Xnojit -Xbootclasspath/p:$APP_HOME/aot/core.jar -classpath:$APP_HOME/aot/main.jar:
                                                                          $APP_HOME/aot/util.jar ...
```

Where core.jar is an IBM-provided file. main.jar and util.jar are your application precompiled files.

Ahead-of-time options

The definitions for the ahead-of-time options.

Purpose

No option specified:
Runs with the interpreter and dynamically compiled code. If AOT code is discovered, it is not used. Instead, it is dynamically compiled as required. This is particularly useful for non-real time and some real-time applications. This option provides optimal performance and throughput but can suffer non-deterministic delays at runtime when compilation occurs.

-Xjit: This option is the same as default.

-Xint: Runs the interpreter only, ignores code written for AOT that might be found in a precompiled jar file, and does not run the dynamic compiler. This mode is not often required, other than for debugging problems that you suspect are related to compilation or for very short batch applications that do not derive benefit from compilation.

-Xnojit:
Runs the interpreter and uses code written for AOT if it is found in a precompiled jar file. It does not run the dynamic compiler. This mode works well for some real-time applications where you want to ensure that no non-deterministic delays occur at runtime because of compilation. Code written for AOT can only be used when running with the **-Xrealtime** option. It is not supported when running in a standard JVM, that is, **-Xrealtime** is not specified.

Example

```
java -Xrealtime -Xnojit outputtest.jar.
```

jxeinajar utility

The jxeinajar utility is used to precompile Java programs for use in WebSphere Real Time.

Purpose

By default, jxeinajar searches for files with a .jar or .zip extension in the current working directory and all its subdirectories.

You can change the input directory by specifying the **-searchPath** option. The search for input files can be extended to include subdirectories of the search path by specifying the **-recurse** option. The optimized jar files written to the **-outPath** directory tree have the same relative structure as the input jar files. You can specify individual jar files as input to jxeinajar.

jxeinajar

Parameters

-help | -?
Shows help.

-Xrealtime
Specifies that the bootclass path that is used for processing contains real-time classes. This option must be supplied on the command line. This option is ignored if it is in the options file.

-outPath
Specifies the root directory other than the current directory or a subdirectory of current directory where the JXEs are generated. The path name can be relative and this option is mandatory.

-[no]logo
Shows the copyright message. **-logo** is a default.

-[no]verify
Verifies the bytecode in the source file. **-verify** is a default.

-searchPath
Is the path where jxeinajar searches for input files with a .jar or .zip extension in the working directory. The default is the current working directory.

-[no]recurse
-recurse extends the search for files with an extension of .jar or .zip to subdirectories of the search path. **-norecurse** is the default.

-noisy
Prints out progress messages for each class in each jar.

-verbose
Prints out progress messages for each jar. **-verbose** is a default.

-quiet
Suppresses all progress messages.

-[no]precompile
Requests precompilation (or no precompilation) of all methods.

-[no]precompileMethod*xxx*
Requests precompilation (or no precompilation) of all methods.

-optFile
Specifies an options file that contains one or more jxeinajar options. An options file can contain any option except another **-optFile** option. The format of the options file is a multiline string of options

[jar file]*
Is a list of specific jar files to process. If no files are specified, all files in the searchPath are converted.

Sample

This example processes the main.jar and util.zip input files, generating the optimized files app/aot/main.jar and app/lib/util.zip. debug.jar is not processed because the **-recurse** flag has not been specified.

```
app/lib/main.jar
app/lib/ext/debug.jar
app/lib/util.zip

cd app
jxeinajar -Xrealtime -searchPath lib -outPath aot
```

Prepending locations to the Java bootclasspath

When precompiling jar files shipped by IBM, to ensure that the precompiled versions of the jar files are loaded instead of the originals, the precompiled jar files must be prepended to the bootclasspath. To prepend a jar file to the bootclasspath use the **-Xbootclasspath/p** <:directories and zip/jar files separated by a colon> option to the Java runtime.

There are also **-Xbootclasspath:** and **-Xbootclasspath/a:** options to the Java runtime, for setting and appending to the bootclasspath respectively, but they are less commonly used and will not ensure that a precompiled jar file is used.

```
java -Xrealtime -Xnojit -Xbootclasspath/p:$APP_HOME/aot/core.jar
                -classpath:$APP_HOME/aot/main.jar:$APP_HOME/aot/util.jar ...
```

Where core.jar is an IBM-provided file. main.jar and util.jar are your application precompiled files.

Return codes from jxeinajar

0 JAR correctly processed.
<10 JIT error and warnings.
11-20 jxeinajar errors.
>100 jar2jxe errors.

Return codes ≤ 5 are considered warnings and processing continues. If multiple warnings or errors are encountered in a single run, the highest value is returned.

To precompile only methods that are used by the application:

1. Run the application with:
   ```
   java -Xjit:verbose={precompile},vlog=optFile
   ```
   to profile the application and store the list of methods to precompile in optFile.
2. Precompile your application using the precompileOpts files generated from the profiling run:
   ```
   jxeinajar -Xrealtime -outPath aot -optFile vlog
   ```

Specific return codes

JIT error codes:

Compilation Failure
1 Compiler ran out of memory trying to perform the compilation of the given method.

Compilation Restriction ILNodes
2 The number of internal IL instructions for this method is too large.

Compilation Restriction RecDepth
3 The stack depth for an optimization phase has exceeded this stack memory.

Compilation Restricted Method
4 Attempt to compile a JNI or abstract method.

Compilation Excessive Complexity
5 Method is too large to compile.

Note: jxeinajar error codes caused by signals have values greater than 128. jxeinajar handles only UNIX signals, such as SIGSEGV, SIGBUS, SIGILL, SIGFPE, SIGTRAP. If one of these signals is intercepted by the jxeinajar command, the return code is 160. For any other signal, the error code is equal to the signal number, + 128, for example, Ctrl-C returns 130.

Known limitation

There is currently a limitation when carrying out ahead-of-time compilation of invalid class files. If you run jxeinajar against a jar file that contains invalid class files (and compilation is requested, that is, you have not specified **-noprecompile**), the processing of the jar file may fail with a return code of 160. As a result jxeinajar terminates without processing the rest of the jar file and therefore fails to produce a new jar file.

As a workaround, you can determine which class file is causing the problem by re-running jxeinajar with the **-noisy** option to determine which class was being compiled at the point of failure. You can then remove the invalid class from the jar file, or update it with a valid class file. Running jxeinajar against the jar file should then complete successfully (assuming it does not contain any additional invalid class files).

Standard options

The definitions for the standard options.

-agentlib:<*libname*>**[=**<*options*>**]**
Loads native agent library <*libname*>; for example **-agentlib:hprof**. For more information, specify **-agentlib:jdwp=help** and **-agentlib:hprof=help** on the command line.

-agentpath:*libname***[=**<*options*>**]**
Loads native agent library by full pathname.

-assert
Prints help on assert-related options.

-cp or -classpath <*directories and zip or jar files separated by* **:**>
Sets the search path for application classes and resources. If **-classpath** and **-cp** are not used and **CLASSPATH** is not set, the user classpath is, by default, the current directory (.).

-D<property_name>=<*value*>
Sets a system property.

-help or -?
Prints a usage message.

-javaagent:<*jarpath*>**[=**<*options*>**]**
Loads Java programming language agent. For more information, see the java.lang.instrument API documentation.

-jre-restrict-search
Includes user private JREs in the version search.

-no-jre-restrict-search
Excludes user private JREs in the version search.

-showversion
Prints product version and continues.

-verbose:[*class***,***gc***,***dynload***,***sizes***,***stack***,,***jni***]**
Enables verbose output.

-verbose:class
Writes an entry to stderr for each class that is loaded.

-verbose:gc
See "Using the -verbose:gc option" on page 23.

-verbose:dynload
Provides detailed information as each class is loaded by the JVM, including:
- The class name and package
- For class files that were in a .jar file, the name and directory path of the .jar
- Details of the size of the class and the time taken to load the class

The data is written out to stderr. An example of the output follows:

```
<Loaded java/lang/String from /myjdk/sdk/jre/lib/vm.jar>
<Class size 17258; ROM size 21080; debug size 0>
<Read time 27368 usec; Load time 782 usec; Translate time 927 usec>
```

Note: Jar files that have been processed by jxeinajar do not display their classes. Use **-verbose:class** for information about these classes.

-verbose:sizes
Writes information to stderr describing the amount of memory used for the stacks and heaps in the JVM

-verbose:stack
Writes information to stderr describing Java and C stack usage.

-verbose:jni
Writes information to stderr describing the JNI services called by the application and JVM.

-version
Prints out the version of the standard JVM. When used with the **-Xrealtime** option, it prints out the WebSphere Real Time product version.

-version:*<value>*
Requires the specified version to run.

-X Prints help on nonstandard options.

Nonstandard garbage collection options

These **-X** options are used with garbage collection and are nonstandard and subject to change without notice.

These options are grouped to show those that can be used with WebSphere Real Time, standard JVM, and with bothMetronome Garbage Collector and WebSphere Real Time and IBM(R) 32-bit SDK for Linux(R) on Intel(R) architecture, Java(TM) 2 Technology Edition, Version 5.0.

Metronome Garbage Collector options

The definitions of the Metronome Garbage Collector options.

-Xgc:immortalMemorySize=*size*
Specifies the size of your immortal heap area. The default is 16 MB.

-Xgc:scopedMemoryMaximumSize=*size*
Specifies the size of your scoped memory heap area. The default is 8 MB.

-Xgc:synchronousGCOnOOM | -Xgc:nosynchronousGCOnOOM
One occasion when garbage collection occurs is when the heap runs out of memory. If there is no more free space in the heap, using **-Xgc:synchronousGCOnOOM** stops your application while garbage collection removes unused objects. If free space runs out again, consider decreasing the target utilization to allow garbage collection more time to complete. Setting **-Xgc:nosynchronousGCOnOOM** implies that when heap memory is full your application stops and issues an out-of-memory message. The default is **-Xgc:synchronousGCOnOOM**.

-Xnoclassgc
Disables class garbage collection. This option switches off garbage collection of storage associated with Java classes that are no longer being used by the JVM. The default behavior is **-Xnoclassgc**.

-Xgc:targetUtilization=*N*
Sets the application utilization to N%; the Garbage Collector attempts to use at most (100-N)% of each time interval. Reasonable values are in the range of 50-80%. Applications with low allocation rates might be able to run at 90%. The default is 70%.

This example shows the maximum size of the heap memory is 30 MB. The garbage collector attempts to use 25% of each time interval because the target utilization for the application is 75%.

```
java -Xrealtime -Xmx30m -Xgc:targetUtilization=75 Test
```

-Xgc:threads=*N*
Specifies the number of GC threads to run. The default is one. The maximum value you can specify is the number of processors available to the operating system.

-Xgc:verboseGCCycleTime=N
N is the time in milliseconds that the summaries should be dumped.

Note: The cycle time does not mean that the summary is dumped precisely at that time, but rather when the last GC quanta or heartbeat that passes this time criteria.

-Xmx<*size*>
Specifies the Java heap size. Unlike other garbage collection strategies, the

real-time Metronome GC does not support heap expansion. There is not an initial or maximum heap size option. You can specify only the maximum heap size.

-Xthr:metronomeAlarm=os*xx*
Controls the priority that the Metronome Garbage Collector alarm thread runs at.

where *xx* is a number from 11 to 89 that specifies the priority the metronome alarm thread should run at. Care should be taken in modifying the OS priority that the alarm thread runs at. If you specify an OS priority lower than that of any realtime thread, you will experience OutOfMemory errors because the Garbage Collector ends up running at a lower priority than realtime threads allocating garbage. The default Metronome Garbage Collector alarm thread runs at an OS priority of 89.

Garbage collection options

The options **-Xdisableexplicitgc**, **-Xnoclassgc**, **-Xgcthreads**, and **-Xverbosegclog:** are the **only** options supported for use with WebSphere Real Time and IBM 32-bit SDK for Linux(R) on Intel(R) architecture, Java(TM) 2 Technology Edition, Version 5.0.

For options that take <*size*> parameter, you should suffix the number with ″k″ or ″K″ to indicate kilobytes, ″m″ or ″M″ to indicate megabytes, or ″g″ or ″G″ to indicate gigabytes.

-Xdisableexplicitgc
Signals to the VM that calls to System.gc() should have no effect. By default, calls to System.gc() trigger a garbage collection.

-Xverbosegclog:<*path to file***><***filename***>[X, Y]**
Causes verboseGC output to be written to the specified file. If the file exists, it is overwritten. Otherwise, if an existing file cannot be opened or a new file cannot be created, the output is redirected to stderr. If you specify the arguments X and Y (both are integers) the verboseGC output is redirected to X number of files, each containing Y number of gc cycles worth of verboseGC output. These files have the form *filename1*, *filename2*, and so on. By default, no verbose garbage collection logging occurs.

-Xgc:threads<*N*>
Specifies the number of GC threads to run. The default is one. The maximum value you can specify is the number of processors available to the operating system. This option applies to standard Java. For WebSphere Real Time see **-Xgcthreads** see “Metronome Garbage Collector options” on page 175.

For definition of **-Xnoclassgc** see “Metronome Garbage Collector options” on page 175.

Garbage collection options used with the standard JVM

These nonstandard option can be used with the standard JVM only. They are not supported for use with WebSphere Real Time.

For options that take <*size*> parameter, you should suffix the number with ″k″ or ″K″ to indicate kilobytes, ″m″ or ″M″ to indicate megabytes, or ″g″ or ″G″ to indicate gigabytes.

-Xclassgc
Enables collection of class objects at every garbage collection. By default,

this option is enabled. For Real-time Java, this option is not supported. Classes cannot be collected by the Metronome Garbage Collector.

-Xcompactgc
Compact every Garbage Collector cycle. See also **-Xnocompactgc**. By default, compaction occurs only when triggered internally.

-Xdisableexcessivegc
Disables the throwing of an OutOfMemoryError if excessive time is spent in the GC. By default, this option is off.

-Xdisablestringconstantgc
Prevents strings in the string intern table from being collected. By default, this option is disabled.

-Xenableexcessivegc
If excessive time is spent in the GC, this option returns NULL for an allocate request and thus causes an OutOfMemoryError to be thrown. This action occurs only when the heap has been fully expanded and the time spent is making up at least 95%. This behavior is the default.

-Xenablestringconstantgc
Enables strings from the string intern table to be collected. By default, this option is enabled.

-Xgcpolicy:[optthruput]|[optavgpause]|[gencon]|[metronome]
Controls the behavior of the Garbage Collector.
- The *optthruput* option is the default and delivers very high throughput to applications, but at the cost of occasional pauses.
- The *optavgpause* option reduces the time that is spent in these garbage collection pauses and limits the effect of increasing heap size on the length of the garbage collection pause. Use *optavgpause* if your configuration has a very large heap.
- The *gencon* option requests the combined use of concurrent and generational GC to help minimize the time that is spent in any garbage collection pause.
- The *metronome* option uses the Metronome Garbage Collector. This is the default when using the **-Xrealtime** option.

For more information, see “Specifying garbage collection policy for non Real-Time” on page 211.

-Xgcworkpackets <*number*>
Specifies the total number of work packets available in the global collector. If not specified, the collector allocates a number of packets based on the maximum heap size.

-Xnocompactgc
Disables compaction for the Garbage Collector. See also **-Xcompactgc**. By default, compaction is enabled.

-Xnopartialcompactgc
Disables incremental compaction. See also **-Xpartialcompactgc**. By default, this option is not set, all compactions are full.

-Xpartialcompactgc
Enables partial compaction. See also **-Xnopartialcompactgc**.

-Xcompactexplicitgc
Compact on every call to System.gc(). See also **-Xnocompactexplicitgc**. By default, compaction occurs only when triggered internally.

-Xnocompactexplicitgc
Disables compaction on a call to `System.gc()`. See also **-Xcompactexplicitgc**. By default, compaction is enabled on calls to `System.gc()`.

-Xlp Requests the JVM to allocate the Java heap with large pages. If large pages are not available, the JVM will not start, displaying the error message `GC: system configuration does not support option --> '-Xlp'`. The JVM uses shmget() to allocate large pages for the heap. Large pages are supported by systems running Linux kernels v2.6 or higher, or earlier kernels where large page support has been backported by the distribution. By default, large pages are not used. See the Diagnostics Guide.

-Xmaxe<*size*>
Sets the maximum amount by which the garbage collector expands the heap. Typically, the garbage collector expands the heap when the amount of free space falls below 30% (or the amount specified using **-Xminf**), by the amount required to restore the free space to 30%. The **-Xmaxe** option limits the expansion to the specified value; for example `-Xmaxe10M` limits the expansion to 10 MB. By default, there is no maximum expansion size.

-Xmaxf<*size*>
Specifies the maximum percentage of heap that must be free after a garbage collection. If the free space exceeds this amount, the JVM attempts to shrink the heap. Specify the size as a decimal value in the range 0-1, for example `-Xmaxf0.5` sets the maximum free space to 50%. The default value is 0.6 (60%).

-Xmine<*size*>
Sets the minimum amount by which the Garbage Collector expands the heap. Typically, the garbage collector expands the heap by the amount required to restore the free space to 30% (or the amount specified using **-Xminf**). The **-Xmine** option sets the expansion to be at least the specified value; for example, `-Xmine50M` sets the expansion size to a minimum of 50 MB. By default, the minimum expansion size is 1 MB.

-Xminf<*size*>
Specifies the minimum percentage of heap that should be free after a garbage collection. If the free space falls below this amount, the JVM attempts to expand the heap. Specify the size as a decimal value in the range 0-1; for example, a value of **-minf0.3** requests the minimum free space to be 30% of the heap. By default, the minimum value is 0.3.

-Xmn<*size*>
Sets the initial and maximum size of the new (nursery) heap to the specified value when using -Xgcpolicy:gencon. Equivalent to setting both **-Xmns** and -Xmnx. If you set either **-Xmns** or **-Xmnx**, you cannot set **-Xmn**. If you attempt to set **-Xmn** with either **-Xmns** or **-Xmnx**, the VM will not start, returning an error. By default, **-Xmn** is selected internally according to your system's capability. You can use the **-verbose:sizes** option to find out the values that the VM is currently using.

-Xmns<*size*>
Sets the initial size of the new (nursery) heap to the specified value when using **-Xgcpolicy:gencon**. By default, this option is selected internally according to your system's capability. This option will return an error if you try to use it with **-Xmn**.

-Xmnx<*size*>
Sets the maximum size of the new (nursery) heap to the specified value

when using **-Xgcpolicy:gencon**. By default, this option is selected internally according to your system's capability. This option will return an error if you try to use it with **-Xmn**.

-Xmo<*size*>
Sets the initial and maximum size of the old (tenured) heap to the specified value when using **-Xgcpolicy:gencon**. Equivalent to setting both **-Xmos** and **-Xmox**. If you set either **-Xmos** or **-Xmox**, you cannot set **-Xmo**. If you attempt to set **-Xmo** with either **-Xmos** or **-Xmox**, the VM will not start, returning an error. By default, **-Xmo** is selected internally according to your system's capability. You can use the **-verbose:sizes** option to find out the values that the VM is currently using.

-Xmos<*size*>
Sets the initial size of the old (tenure) heap to the specified value when using **-Xgcpolicy:gencon**. By default, this option is selected internally according to your system's capability. This option will return an error if you try to use it with **-Xmo**.

-Xmox<*size*>
Sets the maximum size of the old (tenure) heap to the specified value when using **-Xgcpolicy:gencon**. By default, this option is selected internally according to your system's capability. This option will return an error if you try to use it with **-Xmo**.

-Xmoi<*size*>
Sets the amount the Java heap is incremented when using **-Xgcpolicy:gencon**. If set to zero, no expansion is allowed. By default, the increment size is calculated on the expansion size, **-Xmine** and **-Xminf**.

-Xmr<*size*>
Sets the size of the Garbage Collection ″remembered set″ when using **-Xgcpolicy:gencon**. This is a list of objects in the old (tenured) heap that have references to objects in the new (nursery) heap. By default, this option is set to 16 kilobytes.

-Xmrx<*size*>
Sets the remembered maximum size setting.

-Xms<*size*>
Sets the initial Java heap size. You can also use **-Xmo**. The default is set internally according to your system's capability.

Other nonstandard options

These **-X** options are nonstandard and subject to change without notice.

For options that take <*size*> parameter, you should suffix the number with ″k″ or ″K″ to indicate kilobytes, ″m″ or ″M″ to indicate megabytes, or ″g″ or ″G″ to indicate gigabytes.

-Xargencoding
Allows you to put Unicode escape sequences in the argument list. This option is set to off by default.

-Xbootclasspath:<*directories and zip or jar files separated by : >*
Sets the search path for bootstrap classes and resources. The default is to search for bootstrap classes and resources within the internal VM directories and .jar files.

-Xbootclasspath/a:<*directories and zip or jar files separated by : >*
Appends the specified directories, zip, or jar files to the end of bootstrap class path. The default is to search for bootstrap classes and resources within the internal VM directories and .jar files.

-Xbootclasspath/p:<*directories and zip or jar files separated by : >*
Prepends the specified directories, zip, or jar files to the front of the bootstrap class path. Do not deploy applications that use the **-Xbootclasspath:** or **-Xbootclasspath/p:** option to override a class in the standard API, because such a deployment would contravene the Java 2 Runtime Environment binary code license. The default is to search for bootstrap classes and resources within the internal VM directories and .jar files.

-Xcheck:jni
Performs additional checks for JNI functions. You can also use **-Xrunjnichk**. By default, no checking is performed.

-Xcheck:nabounds
Performs additional checks for JNI array operations. You can also use **-Xrunjnichk**. By default, no checking is performed.

-Xcodecache<*size*>
Sets the unit size of which memory blocks are allocated to store native code of compiled Java methods. An appropriate size can be chosen for the application being run. By default, this is selected internally according to the CPU architecture and the capability of your system.

-Xconcurrentbackground <*number*>
Specifies the number of low priority background threads attached to assist the mutator threads in concurrent mark. The default is 1.

-Xconcurrentlevel <*number*>
Specifies the allocation ″tax″ rate. It indicates the ratio between the amount of heap allocated and the amount of heap marked. The default is 8.

-Xconmeter:<soa | loa | dynamic>
Determines which area's usage, LOA (Large Object Area) or SOA (Small Object Area), is metered and hence which allocations are taxed during concurrent mark. The allocation tax is applied to the selected area. If **-Xconmeter:dynamic** is specified, the collector dynamically determines the area to meter based on which area is exhausted first. By default, the option is set to **-Xconmeter:soa**.

-Xdbg:<*options*>
Loads debugging libraries to support the remote debugging of applications. Specifying **-Xrunjdwp** provides the same support. By default, the debugging libraries are not loaded, and the VM instance is not enabled for debug.

-Xdbginfo:<*path to symbol file*>
Loads and passes options to the debug information server. By default, the debug information server is disabled.

-Xdebug
Starts the JVM with the debugger enabled. By default, the debugger is disabled.

-Xdisablejavadump
Turns off javadump generation on errors and signals. By default, javadump generation is enabled.

-Xfuture
Turns on strict class-file format checks. Use this flag when you are developing new code because stricter checks will become the default in future releases. By default, strict format checks are disabled.

-Xint Makes the JVM use only the Interpreter, disabling the Just-In-Time (JIT) compiler. By default, the JIT compiler is enabled.

-Xiss<*size*>
Sets the initial Java thread stack size. 2 KB by default.

-Xjit[:<*suboption*>**,***suboption*,**...]**
Enables the JIT. For details of the suboptions, see theDiagnostics Guide. See also **-Xnojit**. By default, the JIT is enabled.

-Xlinenumbers
Displays line numbers in stack traces, for debugging. See also **-Xnolinenumbers**. By default, line numbers are on.

-Xloa Allocates a large object area (LOA). Objects will be allocated in this LOA rather than the SOA. By default, the LOA is enabled for all GC policies except for subpool, where the LOA is not available. See also **-Xnoloa**.

-Xloainitial <*number*>
<number> is between 0 and 0.95, which specifies the initial percentage of the current tenure space allocated to the large object area (LOA). The default is 0.05 or 5%.

-Xloamaximum <*number*>
<number> is between 0 and 0.95, which specifies the maximum percentage of the current tenure space allocated to the large object area(LOA). The default is 0.5 or 50%.

-Xmca<*size*>
Sets the expansion step for the memory allocated to store the RAM portion of loaded classes. Each time more memory is required to store classes in RAM, the allocated memory is increased by this amount. By default, the expansion step is 32 KB.

-Xmco<*size*>
Sets the expansion step for the memory allocated to store the ROM portion of loaded classes. Each time more memory is required to store classes in ROM, the allocated memory is increased by this amount. By default, the expansion step is 128 KB.

-Xmso<*size*>
Sets the C stack size for forked Java threads. By default, this option is set to 32 KB on 32-bit platforms and 256 KB on 64-bit platforms.

-Xmx<*size*>
Sets maximum Java heap size. By default, this option is set internally according to your system's capability.

-Xnoaot
Disables the AOT (Ahead-of-time) compiler. By default, the AOT compiler is enabled.

-Xnojit
Disables the JIT compiler. See also **-Xjit**. By default, the JIT compiler is enabled.

-Xnolinenumbers
Disables the line numbers for debugging. See also **-Xlinenumbers**. By default, line number are on.

-Xnoloa
Prevents allocation of a large object area (LOA). All objects will be allocated in the SOA. By default, the LOA is enabled for all GC policies except for subpool, where the LOA is not available. See also **-Xloa**.

-Xnosigcatch
Disables JVM signal handling code. See also **-Xsigcatch**. By default, signal handling is enabled.

-Xnosigchain
Disables signal handler chaining. See also **-Xsigchain**. By default, the signal handler chaining is enabled.

-Xoptionsfile=<*file*>
Specifies a file that contains JVM options and defines. By default, no option file is used.

-Xoss<*size*>
Sets the Java stack size and C stack size for any thread. This option is provided for compatibility and is equivalent to setting both **-Xss** and **-Xmso** to the specified value.

-Xquickstart
Improves startup time by delaying JIT compilation and optimizations. By default, quickstart is disabled and there is no delay in JIT compilation.

-Xrdbginfo:<*host*>**:**<*port*>
Loads and passes options to the remote debug information server. By default, the remote debug information server is disabled.

-Xrs Reduces the use of operating system signals. By default, the VM makes full use of operating system signals, see the Diagnostics Guide.

-Xrun<*library name*>**[:options]**
Loads helper libraries. To load multiple libraries, specify it more than once on the command line. Examples of these libraries are:

-Xrunhprof[:help] | [:<*option*>**=**<*value*>**, ...]**
Performs heap, CPU, or monitor profiling. For more information, see the Diagnostics Guide.

-Xrunjdwp[:help] | [:<*option*>=<*value*>, ...]
Loads debugging libraries to support the remote debugging of applications. This is the same as **-Xdbg**. For more information, see the Diagnostics Guide.

-Xrunjnichk[:help] | [:<*option*>=<*value*>, ...]
Performs additional checks for JNI functions, to trace errors in native programs that access the JVM using JNI. For more information, see the Diagnostics Guide.

-Xscmx<*size*>**[k | m | g]**
For details of **-Xscmx**, see "Using command-line options for class data sharing" on page 235.

-Xsigcatch
Enables VM signal handling code. See also **-Xnosigcatch**. By default, signal handling is enabled

-Xsigchain
Enables signal handler chaining. See also **-Xnosigchain**. By default, signal handler chaining is enabled.

-Xsoftrefthreshold<*number*>
Sets the number of GCs after which a soft reference will be cleared if its referent has not been marked. The default is 3, meaning that on the third GC where the referent is not marked the soft reference will be cleared.

-Xss<*size*>
Sets the maximum Java stack size for any thread. By default, this option is set to 256 KB. For more information, see the Diagnostics Guide.

-Xthr:<*options*>
Sets the threading options.

-Xverify
Enables strict class checking for every class that is loaded. By default, strict class checking is disabled.

-Xverify:none
Disables strict class checking. By default, strict class checking is disabled.

WebSphere Real Time class libraries

A reference to the Java class libraries that are used by WebSphere Real Time.

The Java class libraries that are used by WebSphere Real Time are described in http://www.rtsj.org/specjavadoc101b/book_index.html.

Running with TCK

If you are running the Real-Time Specification for Java (RTSJ) Technology Compatibility Kit (TCK) with WebSphere Real Time, you should include demo/realtime/TCKibm.jar in the classpath in order for tests to be completed successfully.

TCKibm.jar includes the class **VibmcorProcessorLock** which is IBM's extension to the TCK.ProcessorLock class. This class provides uniprocessor behavior that is required in a small set of TCK tests. For more information on the TCK.ProcessorLock class and vendor specific extensions to this class, see the README file that is included with the TCK distribution.

Chapter 13. Developing WebSphere Real Time applications using Eclipse

Using Eclipse provides you with a fully-featured IDE when developing your real-time applications.

If this is the first time that you have used the Eclipse application development environment to develop real-time applications, use this procedure to configure your environment.

WebSphere Real Time supplies the standard Sun javac compiler. There are no restrictions on which compiler you use, but it must produce valid Java 5.0 class files. However, the javax.realtime.* Java classes have to be on the build path.

To develop your applications on Eclipse, follow these instructions:

1. Download Eclipse from http://www.eclipse.org/downloads/. It is recommended that you use Eclipse 3.1.2 for correct Java 5.0 compilation.
2. Download IBM SDK and Runtime Environment for Linux platforms, Java 2 Technology Edition, Version 5.0 compliant JVM for running Eclipse.
3. Extract this file, /opt/ibm/java2-i386-50/jre/bin/realtime/jre/bin/realtime//jclSC150/realtime.jar, from the WebSphere Real Time package.
4. Open Eclipse and create a project. Click **File** → **New**. Select **Java project** from the **New Project** panel.

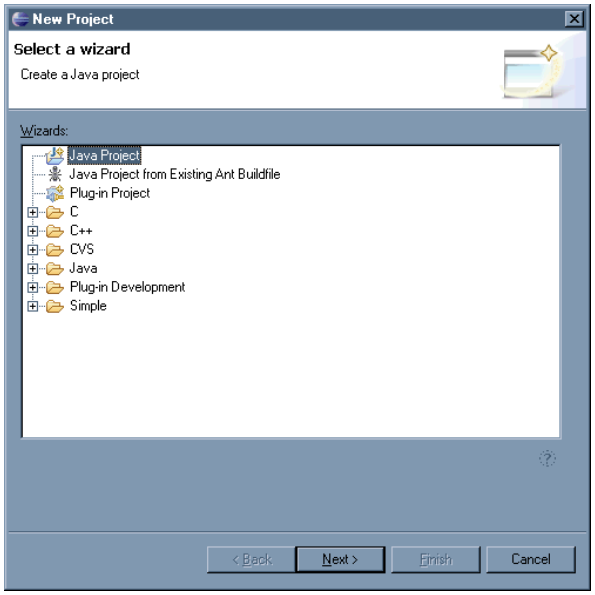

Figure 15. New Project panel

5. Click **Next** to display the **New Java Project** panel.

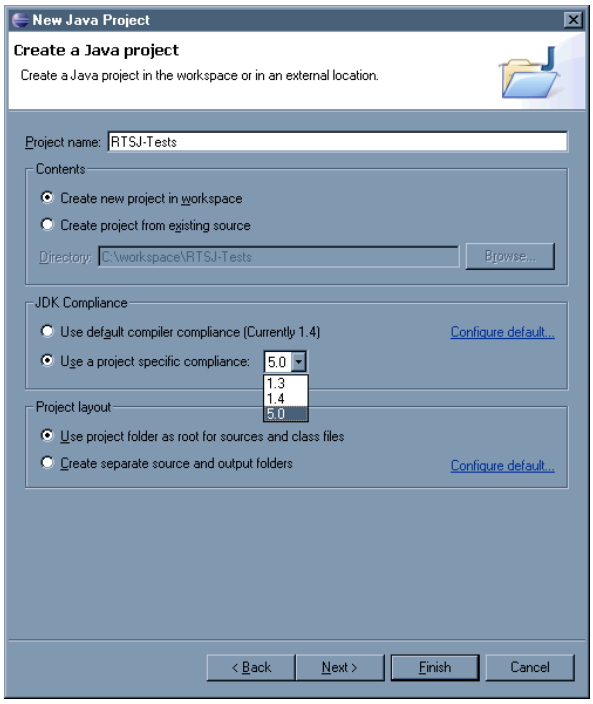

Figure 16. New Java Project panel

a. Enter a project name, for example, RTSJ-Tests.
b. Check that the JDK compiler is set to 5.0.

6. Click **Finish**.
7. Create a working directory and import the /opt/ibm/java2-i386-50/jre/bin/realtime/jre/bin/realtime//jclSC150/realtime.jar file.
8. Click **File** → **New** → **Folder** to open the **New Folder** panel. Enter a new folder name, for example, *deplib*.

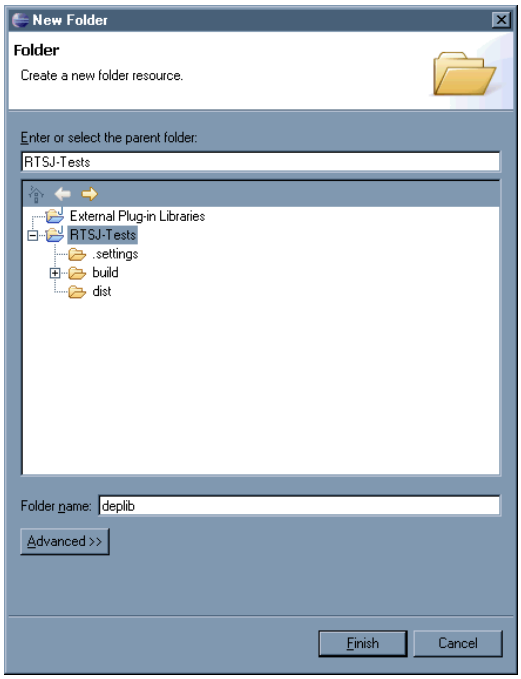

Figure 17. New Folder panel

9. Click **Finish**.
10. To import your realtime.jar file, click **File** → **Import** to open the **Import** panel.

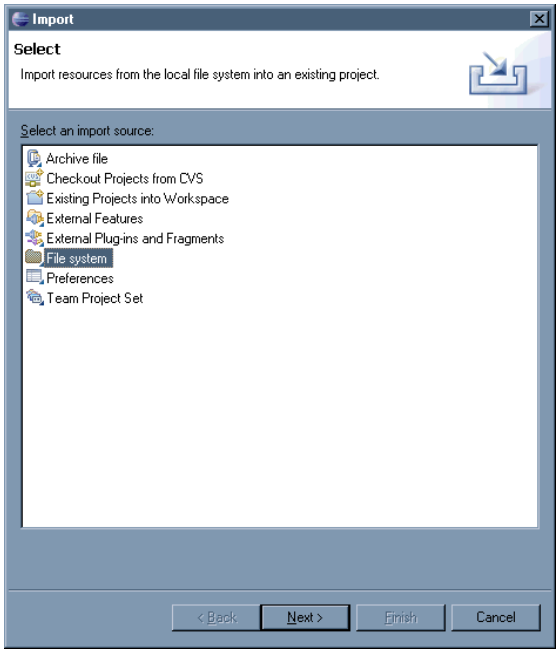

Figure 18. Import - Select panel

11. Click **File System** and click **Next**.
12. Open the /opt/ibm/java2-i386-50/jre/bin/realtime/jclSC150 directory on the filesystem where the JVM was unpacked.

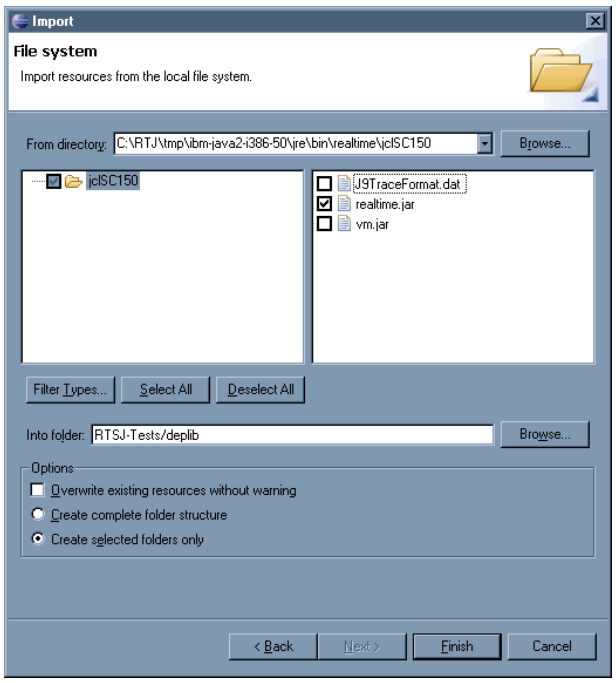

Figure 19. Import - file system panel

13. Click **Finish**.
14. Add the jar file to the library path. Right-click your project and click **Properties** to open the **Properties** panel.

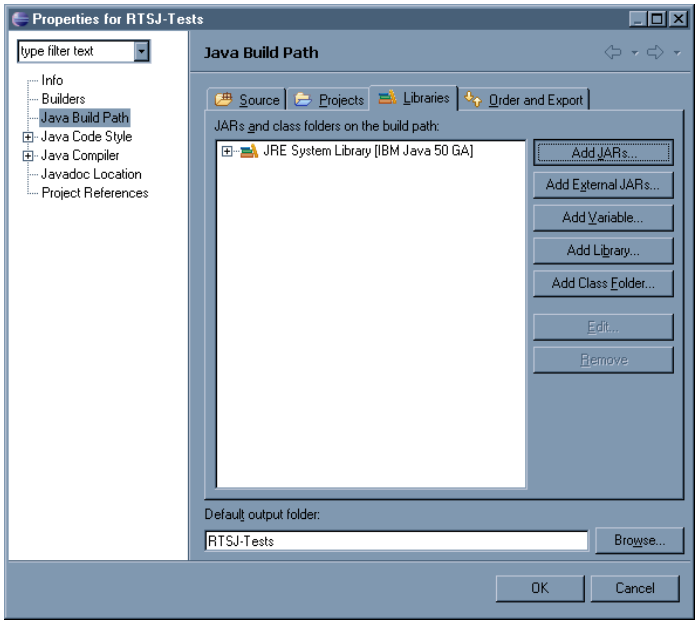

Figure 20. Properties panel

15. Click **Java Build Path** and the **Libraries** tab, as shown in Figure 20. Click **Add Jars**.
16. Click **realtime.jar** under your project directory. Click **OK**.

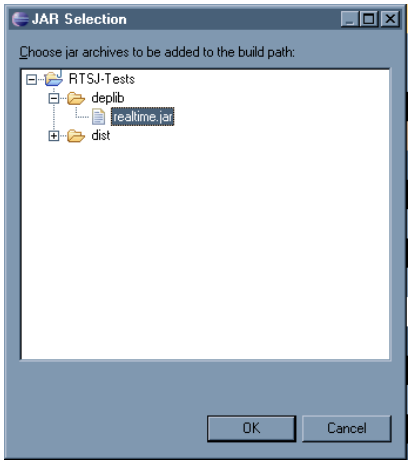

Figure 21. JAR Selection panel

If this procedure was successful, your properties panel will look like Figure 22.

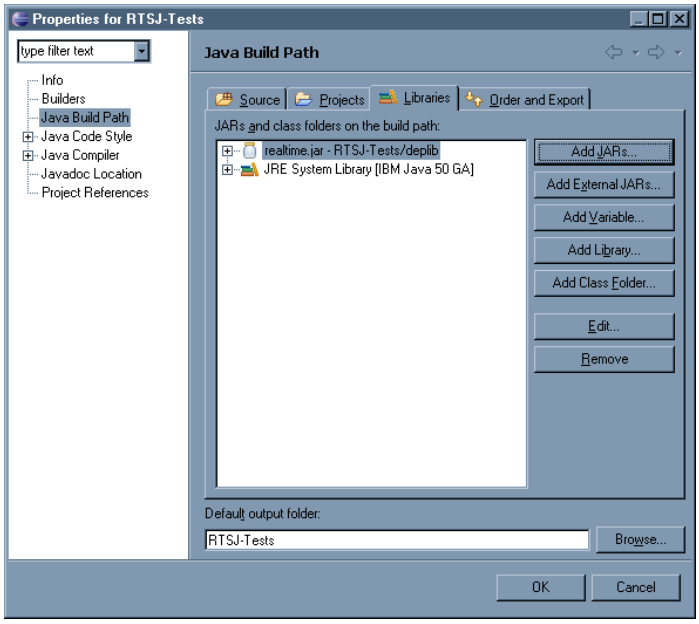

Figure 22. Properties panel

Eclipse can use the realtime.src.jar to present additional information on the RTSJ classes. To do this, open the properties window (see Figure 23 on page 190) for the imported realtime.jar file, click **Java Source Attachment** and enter in **Location path:** the location of the realtime.src.jar file.

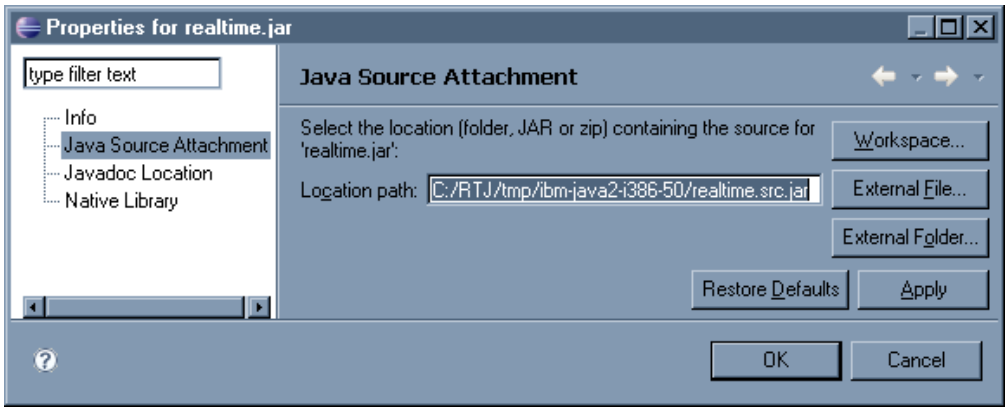

Figure 23. Properties for realtime.src.jar

In addition to build with ant and Eclipse you have to have realtime.jar in your classpath in your ant build script, for example,

```
<property name="rtsj.src" location="." />
<property name="rtsj.deplib" location="deplib" />
<property name="rtsj.jar.dir" location="build/rtsj-jar.dir" />

<!-- Generate .class files for this package -->
<target name="compile" depends="init">
<javac destdir="${rtsj.jar.dir}"
srcdir="${rtsj.src}"
target="1.5"
classpath="${rtsj.deplib}/realtime.jar:${rtsj.src}"
debug="true"/>
</target>
```

This is only a part of an ant build script.

Debugging your applications

Using the Eclipse Application developer you can debug your applications either locally or remotely.

To debug your real-time application remotely, the JVM being debugged requires the following option.
`-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=10100`

1. In the Linux environment where your application is running, enter:

   ```
   java -Xrealtime -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=10100
   ```

   where:
   - `server=y` indicates that the JVM is accepting connections from debuggers.
   - `suspend=y` makes the JVM wait for a debugger to attach before running.
   - `address=10100` is the port number to which the debugger should attach to the JVM. This number should normally be above 1024.

   The JVM outputs the following message:

   ```
   Listening for transport dt_socket at address: 10100
   ```

2. Open your application in Eclipse and select **Debug**.

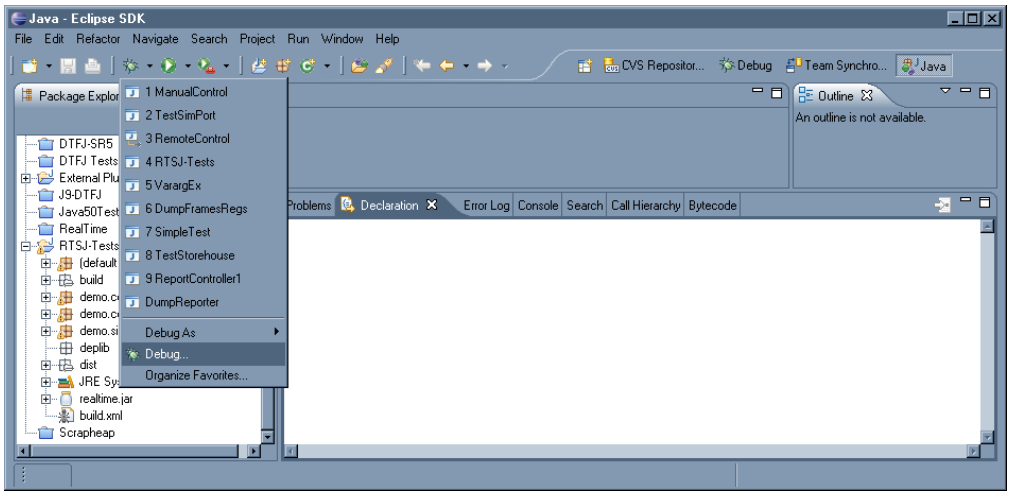

Figure 24. Java - Eclipse panel

3. A new configuration for debugging remote applications should be created. You need only to create one if an application in the same project is run, and is listening on the same port for each run.

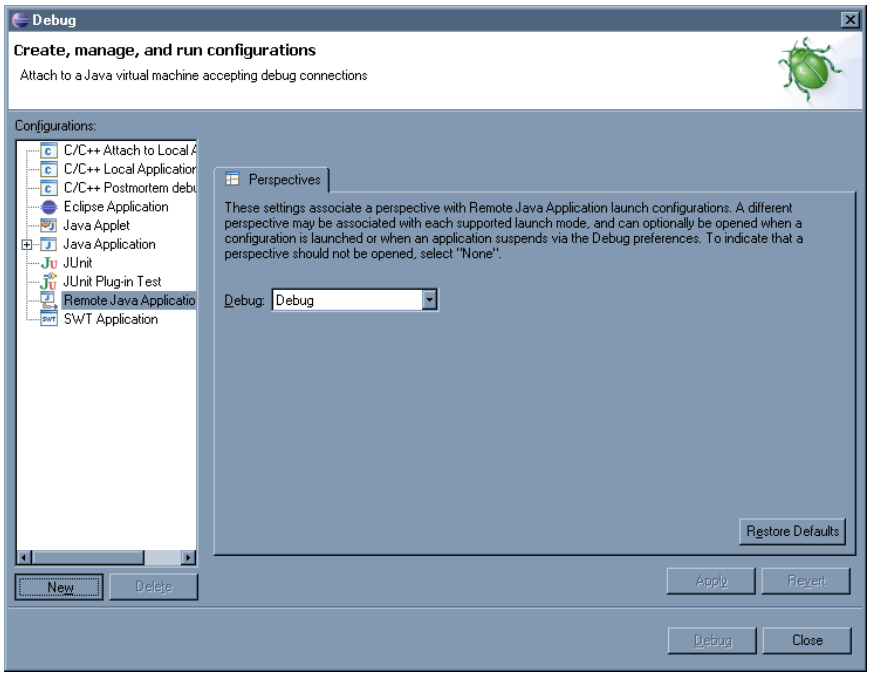

Figure 25. Debug panel

4. When you have created the configuration, fill in the name of the Configurations (RemoteApp in this case), the name of the project that contains the application you are debugging, the *hostname* of the machine where the application is running, and the port number you passed in the **-agentlib** options.

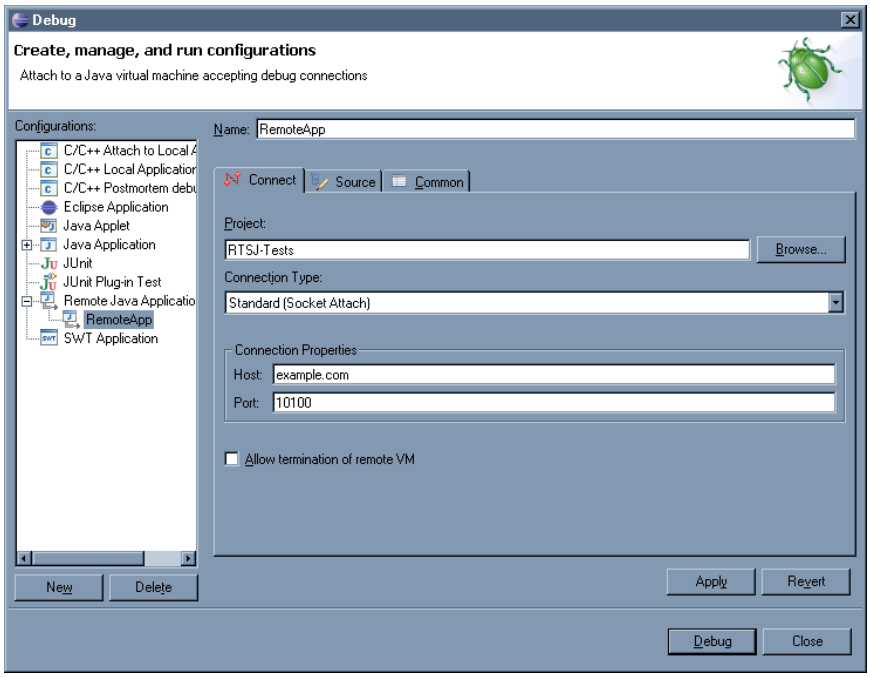

Figure 26. Debug panel

5. Click **Debug** to start the debugging session. The **Debug** perspective should be open for you to view the state of the remotely debugged JVM.

Appendix A. User Guide

IBM SDK and Runtime Environment for Linux platforms, Java 2 Technology Edition, Version 5.0, User Guide

Preface

This User Guide provides general information about the IBM(R) SDK and Runtime Environment for Linux(TM) platforms, Java(TM) 2 Technology Edition, Version 5.0 and specific information about any differences in the IBM implementation compared with the Sun implementation. Read this User Guide in conjunction with the more extensive documentation on the Sun Web site: http://java.sun.com.

For the list of distributions against which the SDK and Runtime Environment for Linux have been tested, see: http://www-106.ibm.com/developerworks/java/jdk/linux/tested.html.

The Diagnostics Guide provides more detailed information about the IBM Virtual Machine for Java.

This User Guide is part of a release and is applicable only to that particular release. Make sure that you have the User Guide appropriate to the release you are using.

The terms ″Runtime Environment″ and ″Java Virtual Machine″ are used interchangeably throughout this User Guide.

Overview

The IBM SDK is a development environment for writing and running applets and applications that conform to the Java 5 Core Application Program Interface (API).

The SDK includes the Runtime Environment for Linux, which enables you only to run Java applications. If you have installed the SDK, the Runtime Environment is included.

The Runtime Environment contains the Java Virtual Machine and supporting files including non-debuggable .so files and class files. The Runtime Environment contains only a subset of the classes that are found in the SDK and allows you to support a Java program at runtime but does not allow you to compile Java programs. The Runtime Environment for Linux does not include any of the development tools, such as **appletviewer** or the Java compiler (**javac**), or classes that are only for development systems.

In addition, for IA32, PPC32/PPC64, and AMD64/EM64T platforms, the Java Communications application programming interface (API) package is provided for use with the Runtime Environment for Linux. You can find information about it in “Using the Java Communications API (JavaComm)” on page 227.

Version compatibility

In general, any applet or application that ran with a previous version of the SDK should run correctly with the IBM SDK for Linux, v5.0. Classes compiled with this release are not guaranteed to work on previous releases.

For information on compatibility issues between release, see the Sun web site at:

http://java.sun.com/j2se/5.0/compatibility.html

http://java.sun.com/j2se/1.4/compatibility.html

http://java.sun.com/j2se/1.3/compatibility.html

There is a possible compatibility consideration if you are using the SDK as part of another product, for example WAS, and you upgrade from a previous level of the SDK, perhaps v1.4.2. If you serialized the classes with a previous level of the SDK, the classes might not be compatible. However, classes are compatible between service refreshes.

Migrating from other IBM JVMs

From Version 1.4.2 for AMD64/EM64T and Version 5 for the other Linux platforms, the IBM Runtime Environment for Linux contains new versions of the IBM Virtual Machine for Java and the Just-In-Time (JIT) compiler.

If you are migrating from an older IBM Runtime Environment, note that:

- To remain compatible with Version 1.4.2, the JVM shared library libjvm.so is installed in both jre/bin/j9vm and jre/bin/classic. Set the **LIBPATH** environment variable to use the JVM shared library in jre/bin/classic when writing native applications using the JNI invocation interface.
- The JVM Monitoring Interface (JVMMI) is no longer available. You must rewrite applications that used that API. You are recommended to use the JVM Tool Interface (JVMTI) instead. The JVMTI is not functionally the equivalent of JVMMI. For information about JVMTI, see http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/ and the Diagnostics Guide.
- The new implementation of JNI conforms to the JNI specification, but differs from the old implementation. It returns copies of objects rather than pinning the objects. This difference can expose errors in JNI application code. For information about debugging JNI code, see **-Xcheck:jni** in "Nonstandard options" on page 203.
- The format and content of garbage collector verbose logs obtained using **-verbose:gc** have changed. The data is now formatted as XML. The data content reflects the changes to the implementation of garbage collection in the JVM, and most of the statistics that are output have changed. You must change any programs that process the verbose GC output so that they will work with the new format and data. See the Diagnostics Guide for an example of the new verbose GC data.
- SDK 1.4 versions of the IBM JRE shipped with JVM specific classes in a file called core.jar. From Java 5.0 onwards, these are shipped in a file called vm.jar.
- Earlier versions of the IBM JRE shipped with a file called rt.jar in the jre/lib directory. From Java v1.4 onwards, this file has been replaced by multiple JAR files that reside in the jre/lib directory.
- For additional industry compatibility and deprecated API information, see Sun's Java 5 Compatibility Documentation and Sun's Java 5 Deprecated API List.

- Tracing class dependencies, invoked using **-verbose:Xclassdep**, is not supported. If you specify **-verbose:Xclassdep**, the JVM will issue an error message and will not start.

Contents of the SDK and Runtime Environment

The SDK contains several development tools and a Java Runtime Environment (JRE). This section describes the contents of the SDK tools and the Runtime Environment.

Applications written entirely in Java should have **no** dependencies on the IBM SDK's directory structure (or files in those directories). Any dependency on the SDK's directory structure (or the files in those directories) could result in application portability problems.

The User Guides, Javadoc, and the accompanying license, copyright files, javadoc, and demo directory are the only documentation included in this SDK for Linux. You can view Sun's software documentation by visiting the Sun Web site, or you can download Sun's software documentation package from the Sun Web site: http://java.sun.com.

Runtime Environment tools

A list of classes and tools that you can use with the standard Runtime Environment.

- Core Classes — These are the compiled class files for the platform and must remain zipped for the compiler and interpreter to access them. Do not modify these classes; instead, create subclasses and override where you need to.
- JRE tools — The following tools are part of the Runtime Environment and are in the /opt/ibm/java2-i386-50/jre/bin directory unless otherwise specified.

  java (Java Interpreter)
  Runs Java classes. The Java Interpreter runs programs that are written in the Java programming language.

  javaw (Java Interpreter)
  Runs Java classes in the same way as the **java** command does, but does not use a console window.

  keytool (Key and Certificate Management Tool)
  Manages a keystore (database) of private keys and their associated X.509 certificate chains that authenticate the corresponding public keys.

  policytool (Policy File Creation and Management Tool)
  Creates and modifies the external policy configuration files that define your installation's Java security policy.

  rmid (RMI activation system daemon)
  Starts the activation system daemon so that objects can be registered and activated in a Java virtual machine (JVM).

  tnameserv (Common Object Request Broker Architecture (CORBA) Naming Service)
  Starts the CORBA transient naming service.

  rmiregistry (Java Remote Object Registry)
  Creates and starts a remote object registry on the specified port of the current host.

ikeyman (iKeyman GUI utility)
Allows you to manage keys, certificates, and certificate requests. For more information see the accompanying *Security Guide* and http://www.ibm.com/developerworks/java/jdk/security. The SDK also provides a command-line version of this utility. See "iKeyman command line tool" on page 241 for details.

jextract (Dump extractor)
Converts a system-produced dump into a common format that can be used by jdmpview. For more information, see the Diagnostics Guide.

***(Linux IA 32-bit, PPC32, and PPC64 only)* javaws (Java Web Start)**
Enables the deployment and automatic maintenance of Java applications.

***(Linux IA 32-bit, PPC32, and PPC64 only)* updateSettings.sh (Java Web Start settings)**
Configures your system to use Web Start with your browser. For more information, see "Running Web Start" on page 233. This file is installed in the jre/lib/javaws directory.

SDK tools

A list of tools that you can use with the standard SDK.

The following tools are part of the SDK and are located in the /opt/ibm/java2-i386-50/bin directory:

javac (Java Compiler)
Compiles programs that are written in the Java programming language into bytecodes (compiled Java code).

appletviewer (Java Applet Viewer)
Tests and runs applets outside a Web browser.

jdb (Java Debugger)
Helps debug your Java programs.

javap (Class File Disassembler)
Disassembles compiled files and can print a representation of the bytecodes.

javadoc (Java Documentation Generator)
Generates HTML pages of API documentation from Java source files.

javah (C Header and Stub File Generator)
Enables you to associate native methods with code written in the Java programming language.

jar (Java Archive Tool)
Combines multiple files into a single Java Archive (JAR) file.

jarsigner (JAR Signing and Verification Tool)
Generates signatures for JAR files and verifies the signatures of signed JAR files.

jconsole (JConsole Monitoring and Management Tool - experimental)
Experimental (unsupported) JMX-compliant graphical tool for monitoring local and remote JVMs.

native2ascii (Native-To-ASCII Converter)
Converts a native encoding file to an ASCII file that contains characters encoded in either Latin-1 or Unicode, or both.

rmic (Java Remote Method Invocation (RMI) Stub Converter)
Generates stubs, skeletons, and ties for remote objects. Includes RMI over Internet Inter-ORB Protocol (RMI-IIOP) support.

java-rmi.cgi (HTTP-to-CGI request forward tool)
Accepts RMI-over-HTTP requests and forwards them to an RMI server listening on any port.

idlj (IDL to Java Compiler)
Generates Java bindings from a given IDL file.

serialver (Serial Version Command)
Returns the serialVersionUID for one or more classes in a format that is suitable for copying into an evolving class.

extcheck (Extcheck utility)
Detects version conflicts between a target jar file and currently-installed extension jar files.

jdmpview (Cross-platform dump formatter)
A tool that allows you to analyze dumps. For more information, see the Diagnostics Guide.

***(Linux IA 32-bit, PPC32, and PPC64 only)* HtmlConverter (Java Plug-in HTML Converter)**
Converts an HTML page that contains applets to a format that can use the Java Plug-in.

Include Files
C headers for JNI programs.

Demos
The demo directory contains a number of subdirectories containing sample source code, demos, applications, and applets that you can use.

copyright
Copyright notice for the SDK for Linux software.

License
The License file, /opt/ibm/java2-i386-50/docs/<*locale*>/license_<*locale*>.html, contains the license agreement for the SDK for Linux software (where <*locale*> is the name of your locale, for example en). To view or print the license agreement, open the file in a Web browser.

fixes.lst
A text file that describes any defects that are fixed after the initial release of this version.

Installing and configuring the SDK and Runtime Environment

You can install the IBM Java SDK and Runtime Environment from either an RPM or a .tgz file. Unless you want to allow all the users on the machine to access this Java installation, use .tgz installation method. If you do not have root access, use the .tgz file.

If you install using an RPM file, the Java files are installed in /opt/ibm/java2-i386-50/. The examples in this guide assume that you have installed Java in this directory.

Upgrading the SDK

If you are upgrading the SDK from a previous release, back up all the configuration files and security policy files before you start the upgrade.

After the upgrade, you might have to restore or reconfigure these files because they might have been overwritten during the upgrade process. Check the syntax of the new files before restoring the original files because the format or options for the files might have changed.

Installing on Red Hat Enterprise Linux (RHEL) 4 or 5

The SDK depends on shared libraries that are not installed by default for Red Hat Enterprise Linux (RHEL).

The RPMs that contain these libraries are:

- `compat-libstdc++-33-3.2.3` and `xorg-x11-deprecated-libs-6.8.1` (Intel Architecture, PPC, and PPC64)
- `compat-libstdc++-295-2.95.3` and `xorg-x11-deprecated-libs-6.8.1` (zSeries)

To include these libraries during RHEL 4 or 5 installation:

1. When you reach the **Package Defaults** screen, select **Customize the set of packages to be installed**.
2. At the **Package Group Selection** screen, under **X Windows System**, choose **Details** and make sure that you have selected `xorg-x11-deprecated-libs`.
3. Under the **Development** options, select **Legacy Software Development**.

Running Java with SELinux on RHEL 5

To run the IBM SDK for Java on Red Hat Enterprise Linux Version 5 with SELinux enabled, Java must be installed in the default directory. If Java is not installed in the default directory, you must disable SELinux or add SELinux exceptions for Java in the installed location.

For more information about SELinux, see http://www.redhat.com/magazine/006apr05/features/selinux/

Installing a 32-bit SDK on 64-bit architecture

To run the SDK, you must install the correct versions of all libraries required by the SDK, either 32- or 64-bit.

In RHEL4, the 64-bit versions of the packages are available in the **Compatibility Arch Support** package group.

You can use the RPM tool to check which versions of the packages you have installed by adding the option `--queryformat "%{NAME}.%{ARCH}\n"` to your RPM command. For example:

```
/home/username : rpm --queryformat "%{NAME}.%{ARCH}\n" -q libstdc++
libstdc++.x86_64
libstdc++.i386
```

Installing from an RPM file

A procedure for installing from an RPM file.

1. Open a shell prompt, making sure you are root.
2. At a shell prompt, type `rpm -ivh <RPM file>`. For example:

```
rpm -ivh ibm-ws-rt-i386-sdk-1.0-1.0.i386.rpm
```

or

```
rpm -ivh ibm-ws-rt-i386-jre-1.0-1.0.i386.rpm
```

Installing from a .tgz file

A procedure for installing from a .tgz file.

1. Create a directory to store the Java package files. The examples in this guide assume that you have installed in /opt/ibm/java2-i386-50/. In the rest of the guide, replace /opt/ibm/java2-i386-50/ with the directory in which you installed Java.
2. At a shell prompt, type `tar -zxvf <.tgz file>`.

   ```
   tar -zxvf ibm-ws-rt-sdk-1.0-1.0-linux-i386.tgz
   ```

   or

   ```
   tar -zxvf ibm-ws-rt-jre-1.0-1.0-linux-i386.tgz
   ```

Configuring the SDK and Runtime Environment for Linux

Inconsistencies in the font encodings on Red Hat Advanced Server

Note: ***(For Linux IA 32-bit Chinese users only)*** Because of inconsistencies in the font encodings on Red Hat Advanced Server, when you install for an environment in which you want Chinese to be the default language, it is better to install with a default language of English and then change to Chinese after the installation is complete. Otherwise, you might find that the Chinese fonts do not display.

Setting the PATH

If you alter the PATH environment variable as described below, you will override any existing Java executables in your path.

After setting the path, you can run a tool by typing its name at a shell prompt with a filename as an argument.

You can specify the path to a tool by typing the path before the name of the tool each time. For example, if the SDK is installed in /opt/ibm/java2-i386-50/bin, you can compile a file named myfile.java by typing the following at a shell prompt:

```
/opt/ibm/java2-i386-50/bin/javac myfile.java
```

To change the **PATH** environment variable:

1. Edit the shell startup file in your home directory (usually .bashrc, depending on your shell) and add the absolute paths to the *PATH* environment variable; for example:

   ```
   export PATH=/opt/ibm/java2-i386-50/bin:/opt/ibm/java2-i386-50/jre/bin:$PATH
   ```
2. Log on again or run the updated shell script to activate the new **PATH** setting.

Setting the class path

The class path tells the SDK tools, such as java, javac, and javadoc, where to find the Java class libraries.

You need to set the class path explicitly only if one of the following applies:

- You require a different library or class file, such as one that you develop, and it is not in the current directory.

- You change the location of the bin and lib directories and they no longer have the same parent directory.
- You plan to develop or run applications using different runtime environments on the same system.

To display the current value of your **CLASSPATH** environment variable, type the following at a shell prompt:

```
echo $CLASSPATH
```

If you develop and run applications that use different runtime environments, including other versions that you have installed separately, you must set the **CLASSPATH** and **PATH** explicitly for each application. If you run multiple applications simultaneously and use different runtime environments, each application must run in its own shell prompt.

If you run only one version of Java at a time, you can use a shell script to switch between the different runtime environments.

Uninstalling the SDK and Runtime Environment for Linux

The process that you use to remove the SDK and Runtime Environment for Linux depends on what type of installation you used.

See Uninstalling the Red Hat Package Manager (RPM) package or Uninstalling the compressed Tape Archive (TAR) package for instructions.

Uninstalling the Red Hat Package Manager (RPM) package

A procedure for uninstalling the Red Hat Package Manager (RPM) package.

To uninstall the SDK or Runtime Environment for Linux if you installed the installable RPM package:

1. To check which RPM packages you have installed, enter: `rpm -qa | grep -i ws-rt`

   You will see a list of any IBM Java packages that you have installed; for example:

   ```
   ibm-ws-rt-i386-jre-1.0-0.0
   ibm-ws-rt-i386-sdk-1.0-0.0
   ```

   This output tells you which packages you can uninstall, using the `rpm -e` command; for example:

   ```
   rpm -e ibm-ws-rt-i386-jre-1.0-0.0
   rpm -e ibm-ws-rt-i386-sdk-1.0-0.0
   ```

   Alternatively, you can use a graphical tool such as kpackage or yast2
2. Remove from your **PATH** statement the directory in which you installed the SDK and Runtime Environment.
3. If you installed the Java Plug-in, remove the Java Plug-in files from the web browser directory.

Uninstalling the SDK and Runtime Environment for Linux

A list of the steps to remove WebSphere Real Time SDK and Runtime Environment for Linux.

1. Remove the Runtime Environment or Runtime Environment files from the directory in which you installed the SDK or Runtime Environment.
2. Remove from your **PATH** statement the directory in which you installed the SDK or Runtime Environment.

3. Log on again or run the updated shell script to activate the new **PATH** setting.
4. If you installed the Java Plug-in, remove the Java Plug-in files from the web browser directory.

Running Java applications

The java and javaw commands

A brief overview of the java and javaw commands.

Purpose

The java and javaw tools launch a Java application by starting a Java Runtime Environment and loading a specified class.

The javaw command is identical to java, except that javaw has no associated console window. Use javaw when you do not want a command prompt window to appear. The javaw launcher displays a dialog box with error information if a launch fails.

Usage

The JVM searches for the initial class (and other classes that are used) in three sets of locations: the bootstrap classpath, the installed extensions, and the user classpath. The arguments that you specify after the class name or JAR file name are passed to the main function.

The java and javaw commands have the following syntax:

```
java [ options ] <class> [ arguments ... ]
java [ options ] -jar <file.jar> [ arguments ... ]
javaw [ options ] <class> [ arguments ... ]
javaw [ options ] -jar <file.jar> [ arguments ... ]
```

Parameters

[*options***]**
Command line options to be passed to the runtime environment.

<class>
Startup class. The class must contain a main() method.

<file.jar>
Name of the jar file to invoke. It is used only with the **-jar** option. The named JAR file must contain class and resource files for the application, with the startup class indicated by the Main-Class manifest header.

[*arguments ...***]**
Command line argument to be passed to the main() function of the invoked class.

Options

The launcher has a set of standard options that are supported on the current runtime environment and will be supported in future releases. In addition, there is a set of nonstandard options. The default options have been chosen for best general use.

Specifying Java options and system properties

You can specify Java options and system properties in these ways.

In order of precedence, they are:

1. By specifying the option or property on the command line. For example, `java -Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump MyJavaClass`.
2. By creating a file that contains the options, and specifying it on the command line using **-Xoptionsfile=**<*filename*>.
3. By creating an environment variable called **IBM_JAVA_OPTIONS** containing the options. For example:

   ```
   export IBM_JAVA_OPTIONS="-Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump"
   ```

Rightmost options on the command line have precedence over leftmost options; for example, if you specify the options `-Xint -Xjit myClass`, `-Xjit` takes precedence.

Standard options

The definitions for the standard options.

-agentlib:<*libname*>**[=**<*options*>**]**
Load native agent library <*libname*>; for example **-agentlib:hprof**. For more information, specify **-agentlib:jdwp=help** and **-agentlib:hprof=help** on the command line.

-agentpath:*libname***[=**<*options*>**]**
Load native agent library by full pathname.

-assert
Prints help on assert-related options.

-cp <*directories and zip or jar files separated by* **:**>
-classpath <*directories and zip or jar files separated by* **:**>
Sets the search path for application classes and resources. If **-classpath** and **-cp** are not used and the **CLASSPATH** environment variable is not set, the user classpath is, by default, the current directory (.).

-D<*property name*>**=**<*value*>
Sets a system property.

-help or -?
Prints a usage message.

-javaagent:<*jarpath*>**[=**<*options*>**]**
Load Java programming language agent. For more information, see the java.lang.instrument API documentation.

-jre-restrict-search
Include user private JREs in the version search.

-no-jre-restrict-search
Exclude user private JREs in the version search.

-showversion
Prints product version and continues.

-verbose:<*option*>
Enables verbose output. The available options are:

class
Writes an entry to stderr for each class that is loaded.

gc See “Using the -verbose:gc option” on page 23.

jni
Writes information to stderr describing the JNI services called by the application and JVM.

sizes
Writes information to stderr describing the active memory usage settings.

stack
Writes information to stderr describing the Java and C stack usage for each thread.

-version
Prints out the version of the standard JVM. When used in conjunction with the **-Xrealtime** option, it prints out the Websphere Real Time product version.

-version:<*value*>
Require the specified version to run.

-X Prints help on nonstandard options.

Nonstandard options

The **-X** options listed below are nonstandard and subject to change without notice.

For options that take a <*size*> parameter, you should suffix the number with ″k″ or ″K″ to indicate kilobytes, ″m″ or ″M″ to indicate megabytes, or ″g″ or ″G″ to indicate gigabytes.

-Xargencoding
Allows you to put Unicode escape sequences in the argument list. This option is set to off by default.

-Xbootclasspath:<*directories and zip or jar files separated by :*>
Sets the search path for bootstrap classes and resources. The default is to search for bootstrap classes and resources within the internal VM directories and .jar files.

-Xbootclasspath/a:<*directories and zip or jar files separated by :*>
Appends the specified directories, zip, or jar files to the end of bootstrap class path. The default is to search for bootstrap classes and resources within the internal VM directories and .jar files.

-Xbootclasspath/p:<*directories and zip or jar files separated by :*>
Prepends the specified directories, zip, or jar files to the front of the bootstrap class path. Do not deploy applications that use the **-Xbootclasspath:** or **-Xbootclasspath/p:** option to override a class in the standard API, because such a deployment would contravene the Java 2 Runtime Environment binary code license. The default is to search for bootstrap classes and resources within the internal VM directories and .jar files.

-Xcheck:jni
Performs additional checks for JNI functions. By default, no checking is performed. For more information, see the Diagnostics Guide.

-Xcheck:nabounds
Performs additional checks for JNI array operations. You can also use **-Xrunjnichk**. By default, no checking is performed.

-Xclassgc
Enables collection of class objects at every garbage collection. By default, this option is enabled. For Real-time Java, this option is not supported. Classes cannot be collected by the Metronome Garbage Collector.

-Xcodecache<*size*>
Sets the unit size of which memory blocks are allocated to store native code of compiled Java methods. An appropriate size can be chosen for the application being run. By default, this is selected internally according to the CPU architecture and the capability of your system.

-Xcompactexplicitgc
Compact on every call to `System.gc()`. See also **-Xnocompactexplicitgc**. By default, compaction occurs only when triggered internally.

Note: This option should not be used with the metronome garbage collector (**-Xrealtime**).

-Xcompactgc
Compact every Garbage Collector cycle. See also **-Xnocompactgc**. By default, compaction occurs only when triggered internally.

-Xconcurrentbackground<*number*>
Specifies the number of low priority background threads attached to assist the mutator threads in concurrent mark. The default is 1.

-Xconcurrentlevel<*number*>
Specifies the allocation "tax" rate. It indicates the ratio between the amount of heap allocated and the amount of heap marked. The default is 8.

-Xconmeter:<*soa* | *loa* | *dynamic*>
Determines which area's usage, LOA (Large Object Area) or SOA (Small Object Area), is metered and hence which allocations are taxed during concurrent mark. The allocation tax is applied to the selected area. If **-Xconmeter:dynamic** is specified, the collector dynamically determines the area to meter based on which area is exhausted first. By default, the option is set to **-Xconmeter:soa**.

-Xdbg:<*options*>
Loads debugging libraries to support the remote debugging of applications. Specifying **-Xrunjdwp** provides the same support. By default, the debugging libraries are not loaded, and the VM instance is not enabled for debug.

-Xdbginfo:<*path to symbol file*>
Loads and passes options to the debug information server. By default, the debug information server is disabled.

-Xdebug
Starts the JVM with the debugger enabled. By default, the debugger is disabled.

-Xdisableexcessivegc
Disables the throwing of an OutOfMemoryError if excessive time is spent in the GC. By default, this option is off.

-Xdisableexplicitgc
Signals to the VM that calls to System.gc() should have no effect. By default, calls to System.gc() trigger a garbage collection.

-Xdisablestringconstantgc
Prevents strings in the string intern table from being collected. By default, this option is disabled.

-Xdisablejavadump
Turns off Javadump generation on errors and signals. By default, Javadump generation is enabled.

-Xenableexcessivegc
If excessive time is spent in the GC, this option returns NULL for an allocate request and thus causes an OutOfMemoryError to be thrown. This action occurs only when the heap has been fully expanded and the time spent is making up at least 95%. This behavior is the default.

-Xenablestringconstantgc
Enables strings from the string intern table to be collected. By default, this option is enabled.

-Xfuture
Turns on strict class-file format checks. Use this flag when you are developing new code because stricter checks will become the default in future releases. By default, strict format checks are disabled.

-Xgcpolicy:<*optthruput* | *optavgpause* | *gencon*> **(subpool on PPC and zSeries)**
Controls the behavior of the Garbage Collector. See "Garbage collection options" on page 211 for more information.

-Xgcthreads<*number of threads*>
Sets the number of helper threads that are used for parallel operations during garbage collection. By default, the number of threads is set to the number of physical CPUs present -1, with a minimum of 1.

-Xgcworkpackets<*number*>
Specifies the total number of work packets available in the global collector. If not specified, the collector allocates a number of packets based on the maximum heap size.

-Xint
Makes the JVM use only the Interpreter, disabling the Just-In-Time (JIT) compiler. By default, the JIT compiler is enabled.

-Xiss<*size*>
Sets the initial Java thread stack size. 2 KB by default.

-Xjarversion
Outputs information on the version of each jar file in the class path, the boot class path, and the extensions directory. Version information is taken from the Implementation-Version and Build-Level properties in the manifest of the jar.

-Xjit[:<*suboption*>,<*suboption*>...]
Enables the JIT. For details of the suboptions, see the Diagnostics Guide. See also **-Xnojit**. By default, the JIT is enabled.

-Xlinenumbers
Displays line numbers in stack traces, for debugging. See also **-Xnolinenumbers**. By default, line numbers are on.

-Xloa
Allocates a large object area (LOA). Objects will be allocated in this LOA rather than the SOA. By default, the LOA is enabled for all GC policies except for subpool, where the LOA is not available. See also **-Xnoloa**.

-Xloainitial<*percentage*>
<*percentage*> is between 0 and 0.95, which specifies the initial percentage of the current tenure space allocated to the large object area (LOA). The default is 0.05 or 5%.

-Xloamaximum<*percentage*>
<*percentage*> is between 0 and 0.95, which specifies the maximum percentage of the current tenure space allocated to the large object area (LOA). The default is 0.5 or 50%.

-Xlp
Requests the JVM to allocate the Java heap with large pages. If large pages are not available, the JVM will not start, displaying the error message `GC: system configuration does not support option --> '-Xlp'`. The JVM uses shmget() to allocate large pages for the heap. Large pages are supported by systems running Linux kernels v2.6 or higher, or earlier kernels where large page support has been backported by the distribution. By default, large pages are not used. See "Configuring large page memory allocation" on page 222.

Note: This option should not be used with the metronome garbage collector (**-Xrealtime**).

-Xmaxe<*size*>
Sets the maximum amount by which the garbage collector expands the heap. Typically, the garbage collector expands the heap when the amount of free space falls below 30% (or the amount specified using **-Xminf**), by the amount required to restore the free space to 30%. The **-Xmaxe** option limits the expansion to the specified value; for example `-Xmaxe10M` limits the expansion to 10 MB. By default, there is no maximum expansion size.

Note: This option should not be used with the metronome garbage collector (**-Xrealtime**).

-Xmaxf<*size*>
Specifies the maximum percentage of heap that must be free after a garbage collection. If the free space exceeds this amount, the JVM attempts to shrink the heap. Specify the size as a decimal value in the range 0-1, for example `-Xmaxf0.5` sets the maximum free space to 50%. The default value is 0.6 (60%).

Note: This option should not be used with the metronome garbage collector (**-Xrealtime**).

-Xmca<*size*>
Sets the expansion step for the memory allocated to store the RAM portion of loaded classes. Each time more memory is required to store classes in RAM, the allocated memory is increased by this amount. By default, the expansion step is 32 KB.

-Xmco<*size*>
Sets the expansion step for the memory allocated to store the ROM portion of loaded classes. Each time more memory is required to store classes in ROM, the allocated memory is increased by this amount. By default, the expansion step is 128 KB.

-Xmine<*size*>
Sets the minimum amount by which the Garbage Collector expands the heap. Typically, the garbage collector expands the heap by the amount required to restore the free space to 30% (or the amount specified using **-Xminf**). The **-Xmine** option sets the expansion to be at least the specified value; for example, `-Xmine50M` sets the expansion size to a minimum of 50 MB. By default, the minimum expansion size is 1 MB.

Note: This option should not be used with the metronome garbage collector (**-Xrealtime**).

-Xminf<*size*>
Specifies the minimum percentage of heap that should be free after a garbage collection. If the free space falls below this amount, the JVM attempts to expand the heap. Specify the size as a decimal value in the range 0-1; for example, a value of **–Xminf0.3** requests the minimum free space to be 30% of the heap. By default, the minimum value is 0.3.

Note: This option should not be used with the metronome garbage collector (**-Xrealtime**).

-Xmn<*size*>
Sets the initial and maximum size of the new (nursery) heap to the specified value when using **-Xgcpolicy:gencon**. Equivalent to setting both **-Xmns** and **-Xmnx**. If you set either **-Xmns** or **-Xmnx**, you cannot set **-Xmn**. If you attempt to set **-Xmn** with either **-Xmns** or **-Xmnx**, the VM will not start, returning an error. By default, **-Xmn** is selected internally according to your system's capability. You can use the **-verbose:sizes** option to find out the values that the VM is currently using.

Note: This option should not be used with the metronome garbage collector (**-Xrealtime**).

-Xmns<*size*>
Sets the initial size of the new (nursery) heap to the specified value when using **-Xgcpolicy:gencon**. By default, this option is selected internally according to your system's capability. This option will return an error if you try to use it with **-Xmn**.

Note: This option should not be used with the metronome garbage collector (**-Xrealtime**).

-Xmnx<*size*>
Sets the maximum size of the new (nursery) heap to the specified value when using **-Xgcpolicy:gencon**. By default, this option is selected internally according to your system's capability. This option will return an error if you try to use it with **-Xmn**.

Note: This option should not be used with the metronome garbage collector (**-Xrealtime**).

-Xmo<*size*>
Sets the initial and maximum size of the old (tenured) heap to the specified value when using **-Xgcpolicy:gencon**. Equivalent to setting both **-Xmos** and **-Xmox**. If you set either **-Xmos** or **-Xmox**, you cannot set **-Xmo**. If you attempt to set **-Xmo** with either **-Xmos** or **-Xmox**, the VM will not start, returning an error. By default, **-Xmo** is selected internally according to your system's capability. You can use the **-verbose:sizes** option to find out the values that the VM is currently using.

Note: This option should not be used with the metronome garbage collector (**-Xrealtime**).

-Xmos<*size*>
Sets the initial size of the old (tenure) heap to the specified value when using **-Xgcpolicy:gencon**. By default, this option is selected internally according to your system's capability. This option will return an error if you try to use it with **-Xmo**.

Note: This option should not be used with the metronome garbage collector (**-Xrealtime**).

-Xmox<*size*>
Sets the maximum size of the old (tenure) heap to the specified value when using **-Xgcpolicy:gencon**. By default, this option is selected internally according to your system's capability. This option will return an error if you try to use it with **-Xmo**.

Note: This option should not be used with the metronome garbage collector (**-Xrealtime**).

-Xmoi<*size*>
Sets the amount the Java heap is incremented when using **-Xgcpolicy:gencon**. If set to zero, no expansion is allowed. By default, the increment size is calculated on the expansion size, **-Xmine** and **-Xminf**.

Note: This option should not be used with the metronome garbage collector (**-Xrealtime**).

-Xmr<*size*>
Sets the size of the Garbage Collection "remembered set" when using **-Xgcpolicy:gencon**. This is a list of objects in the old (tenured) heap that have references to objects in the new (nursery) heap. By default, this option is set to 16 kilobytes.

Note: This option should not be used with the metronome garbage collector (**-Xrealtime**).

-Xmrx<*size*>
Sets the remembered maximum size setting.

Note: This option should not be used with the metronome garbage collector (**-Xrealtime**).

-Xms<*size*>
Sets the initial Java heap size. You can also use **-Xmo**. The default is set internally according to your system's capability.

Note: This option should not be used with the metronome garbage collector (**-Xrealtime**).

-Xmso<*size*>
Sets the C stack size for forked Java threads. By default, this option is set to 32 KB on 32-bit platforms and 256 KB on 64-bit platforms.

-Xmx<*size*>
Sets maximum Java heap size. By default, this option is set internally according to your system's capability.

-Xnoaot
Disables the AOT (Ahead-of-time) compiler. By default, the AOT compiler is enabled.

-Xnoclassgc
Disables class garbage collection. This option switches off garbage collection of storage associated with Java classes that are no longer being used by the JVM. The default behavior is **-Xclassgc**.

-Xnocompactgc
Disables compaction for the Garbage Collector. See also **-Xcompactgc**. By default, compaction is enabled.

-Xnocompactexplicitgc
Disables compaction on a call to `System.gc()`. See also **-Xcompactexplicitgc**. By default, compaction is enabled on calls to `System.gc()`.

Note: This option should not be used with the metronome garbage collector (**-Xrealtime**).

-Xnojit
Disables the JIT compiler. See also **-Xjit**. By default, the JIT compiler is enabled.

-Xnolinenumbers
Disables the line numbers for debugging. See also **-Xlinenumbers**. By default, line number are on.

-Xnoloa
Prevents allocation of a large object area (LOA). All objects will be allocated in the SOA. By default, the LOA is enabled for all GC policies except for subpool, where the LOA is not available. See also **-Xloa**.

-Xnopartialcompactgc
Disables incremental compaction. See also **-Xpartialcompactgc**. By default, this option is not set, so all compactions are full.

-Xnosigcatch
Disables JVM signal handling code. See also **-Xsigcatch**. By default, signal handling is enabled.

-Xnosigchain
Disables signal handler chaining. See also **-Xsigchain**. By default, the signal handler chaining is enabled.

-Xoptionsfile=<*file*>
Specifies a file that contains JVM options and defines. By default, no option file is used.

-Xoss<*size*>
Sets the Java stack size and C stack size for any thread. This option is provided for compatibility and is equivalent to setting both **-Xss** and **-Xmso** to the specified value.

-Xpartialcompactgc
Enables partial compaction. See also **-Xnopartialcompactgc**.

-Xquickstart
Improves startup time by delaying JIT compilation and optimizations. By default, quickstart is disabled and there is no delay in JIT compilation.

-Xrdbginfo:<*host*>**:**<*port*>
Loads and passes options to the remote debug information server. By default, the remote debug information server is disabled.

-Xrs
Reduces the use of operating system signals. By default, the VM makes full use of operating system signals, see “Signals used by the JVM” on page 214.

-Xrun<*library name*>**[:**<*options*>**]**
Loads helper libraries. To load multiple libraries, specify it more than once on the command line. Examples of these libraries are:

-Xrunhprof[:help] | [:<*option*>=<*value*>, ...]
Performs heap, CPU, or monitor profiling. For more information, see the Diagnostics Guide.

-Xrunjdwp[:help] | [:<*option*>=< *value*>, ...]
Loads debugging libraries to support the remote debugging of applications. This is the same as **-Xdbg**. For more information, see "Debugging Java applications" on page 218.

-Xrunjnichk[:help] | [:<*option*>=<*value*>, ...]
Deprecated, use **-Xcheck:jni**.

-Xscmx<*size*>
For details of **-Xscmx**, see "Using command-line options for class data sharing" on page 235.

-Xsigcatch
Enables VM signal handling code. See also **-Xnosigcatch**. By default, signal handling is enabled.

-Xsigchain
Enables signal handler chaining. See also **-Xnosigchain**. By default, signal handler chaining is enabled.

-Xsoftrefthreshold<*number*>
Sets the number of GCs after which a soft reference will be cleared if its referent has not been marked. The default is 3, meaning that on the third GC where the referent is not marked the soft reference will be cleared.

-Xss<*size*>
Sets the maximum Java stack size for any thread. By default, this option is set to 256 KB. For more information, see Working with floating stacks.

-Xthr:<*options*>
Sets the threading options.

-Xverbosegclog:<*path to file*>**[***X***,***Y***]**
Causes verboseGC output to be written to the specified file. If the file exists, it is overwritten. Otherwise, if an existing file cannot be opened or a new file cannot be created, the output is redirected to stderr. If you specify the arguments X and Y (both are integers) the verboseGC output is redirected to X number of files, each containing Y number of gc cycles worth of verboseGC output. These files have the form *filename1, filename2*, and so on. By default, no verbose garbage collection logging occurs.

-Xverify
Enables strict class checking for every class that is loaded. By default, strict class checking is disabled.

-Xverify:none
Disables strict class checking. By default, strict class checking is disabled.

Obtaining the IBM build and version number

This task describes how to obtain the IBM build and version number for your Java installation.

1. Open a shell prompt.
2. Type the following command:

```
java -version
```

Globalization of the java command

The java command and other Java launcher commands (such as javaw) allow a class name to be specified as any character that is in the character set of the current locale.

You can also specify any Unicode character in the class name and arguments by using Java escape sequences.

To do this, use the **-Xargencoding** command line option.

-Xargencoding
Use argument encoding. To specify a Unicode character, use escape sequences in the form \u####, where # is a hexadecimal digit (0 through 9, A through F).

-Xargencoding:utf8
Use UTF8 encoding.

-Xargencoding:latin
Use ISO8859_1 encoding.

For example, to specify a class called HelloWorld using Unicode encoding for both capital letters, use this command:

```
java -Xargencoding '\u0048ello\u0057orld'
```

The java and javaw commands give translated output messages. These messages differ based on the locale in which Java is running. The detailed error descriptions and other debug information that is returned by java are in English.

Specifying garbage collection policy for non Real-Time

The Garbage Collector manages the memory used by Java and by applications running within the VM. These Garbage Collection policies are not available when the **-Xrealtime** option is specified.

When the Garbage Collector receives a request for storage, unused memory in the heap is set aside in a process called "allocation". The Garbage Collector also checks for areas of memory are no longer referenced, and releases them for reuse. This is known as "collection".

The collection phase can be triggered by a memory allocation fault, which occurs when no space is left for a storage request, or by an explicit System.gc() call.

Garbage collection can significantly affect application performance, so the IBM virtual machine provides various methods of optimizing the way garbage collection is carried out, potentially reducing the effect on your application.

For more detailed information on garbage collection, see the Diagnostics Guide.

Garbage collection options

The **-Xgcpolicy** options control the behavior of the Garbage Collector, It makes trade-offs between throughput of the application and overall system, and the pause times that are caused by garbage collection.

For definitions of **-Xgcpolicy:[optthruput]|[optavgpause]|[gencon]** see "Garbage collection options" on page 176.

Pause time

When an application's attempt to create an object cannot be satisfied immediately from the available space in the heap, the garbage collector is responsible for identifying unreferenced objects (garbage), deleting them, and returning the heap to a state in which the immediate and subsequent allocation requests can be satisfied quickly.

Such garbage collection cycles introduce occasional unexpected pauses in the execution of application code. Because applications grow in size and complexity, and heaps become correspondingly larger, this garbage collection pause time tends to grow in size and significance.

The default garbage collection value, **-Xgcpolicy:optthruput**, delivers very high throughput to applications, but at the cost of these occasional pauses, which can vary from a few milliseconds to many seconds, depending on the size of the heap and the quantity of garbage.

Pause time reduction

The JVM uses two techniques to reduce pause times: Concurrent garbage collection and Generational garbage collection

The **-Xgcpolicy:optavgpause** command-line option requests the use of concurrent garbage collection to reduce significantly the time that is spent in garbage collection pauses. Concurrent GC reduces the pause time by performing some garbage collection activities concurrently with normal program execution to minimize the disruption caused by the collection of the heap. The **-Xgcpolicy:optavgpause** option also limits the effect of increasing the heap size on the length of the garbage collection pause. The **-Xgcpolicy:optavgpause** option is most useful for configurations that have large heaps. With the reduced pause time, you might experience some reduction of throughput to your applications.

During concurrent garbage collection, a significant amount of time is wasted identifying relatively long-lasting objects that cannot then be collected. If the GC concentrates on only the objects that are most likely to be recyclable, you can further reduce pause times for some applications. Generational GC reduces pause times by dividing the heap into two ″generations″: the ″nursery″ and the ″tenure″ areas. Objects are placed in one of these areas depending on their age. The nursery is the smaller of the two and contains younger objects; the tenure is larger and contains older objects. Objects are first allocated to the nursery; if they survive long enough, they are promoted to the tenure area eventually.

Generational GC depends on most objects not lasting long. Generational GC reduces pause times by concentrating the effort to reclaim storage on the nursery because it has the most recyclable space. Rather than occasional but lengthy pause times to collect the entire heap, the nursery is collected more frequently and, if the nursery is small enough, pause times are comparatively short. However, generational GC has the drawback that, over time, the tenure area might become full if too many objects last too long. To minimize the pause time when this situation occurs, use a combination of concurrent GC and generational GC. The **-Xgcpolicy:gencon** option requests the combined use of concurrent and generational GC to help minimize the time that is spent in any garbage collection pause.

Environments with very full heaps

If the Java heap becomes nearly full, and very little garbage can be reclaimed, requests for new objects might not be satisfied quickly because no space is immediately available.

If the heap is operated at near-full capacity, application performance might suffer regardless of which of the above options is used; and, if requests for more heap space continue to be made, the application may receive an OutOfMemoryError, which results in JVM termination if the exception is not caught and handled. At this point the JVM produces a "javadump" diagnostic file. In these conditions, you are recommended either to increase the heap size by using the **-Xmx** option, or to reduce the number of objects in use. For more information, see the Diagnostics Guide.

How the JVM processes signals

When a signal is raised that is of interest to the JVM, a signal handler is called. This signal handler determines whether it has been called for a Java or non-Java thread.

If the signal is for a Java thread, the JVM takes control of the signal handling. If an application handler for this signal is installed and you did not specify the **-Xnosigchain** command-line option, the application handler for this signal is called after the JVM has finished processing.

If the signal is for a non-Java thread, and the application that installed the JVM had previously installed its own handler for the signal, control is given to that handler. Otherwise, if the signal is requested by the JVM or Java application, the signal is ignored or the default action is taken.

For exception and error signals, the JVM either:

- Handles the condition and recovers, or
- Enters a controlled shutdown sequence where it:
  1. Outputs a Javadump, to describe the JVM state at the point of failure
  2. Calls your application's signal handler for that signal
  3. Calls any application-installed abort hook
  4. Performs the necessary cleanup to give a clean shutdown

For information about writing a launcher that specifies the above hooks, see: http://www.ibm.com/developerworks/java/library/i-signalhandling/. This item was written for Java V1.3.1, but still applies to later versions.

For interrupt signals, the JVM also enters a controlled shutdown sequence, but this time it is treated as a normal termination that:

- Calls your application's signal handler for that signal.
- Runs all application shutdown hooks.
- Calls any application-installed exit hook.
- Performs the necessary JVM cleanup.

The shutdown is identical to the shutdown initiated by a call to the Java method `System.exit()`.

Other signals that are used by the JVM are for internal control purposes and do not cause it to terminate. The only control signal of interest is SIGQUIT, which causes a Javadump to be generated.

Signals used by the JVM

The types of signals are Exceptions, Errors, Interrupts, and Controls.

Table 23 below shows the signals that are used by the JVM. The signals are grouped in the table by type or use, as follows:

Exceptions
The operating system synchronously raises an appropriate exception signal whenever a fatal condition occurs.

Errors The JVM raises a SIGABRT if it detects a condition from which it cannot recover.

Interrupts
Interrupt signals are raised asynchronously, from outside a JVM process, to request shutdown.

Controls
Other signals that are used by the JVM for control purposes.

Table 23. Signals used by the JVM

Signal Name	Signal type	Description	Disabled by -Xrs
SIGBUS (7)	Exception	Incorrect access to memory (data misalignment)	Yes
SIGSEGV (11)	Exception	Incorrect access to memory (write to inaccessible memory)	Yes
SIGILL (4)	Exception	Illegal instruction (attempt to invoke an unknown machine instruction)	No
SIGFPE (8)	Exception	Floating point exception (divide by zero)	Yes
SIGABRT	Error	Abnormal termination. The JVM raises this signal whenever it detects a JVM fault.	Yes
SIGINT (2)	Interrupt	Interactive attention (CTRL-C). JVM exits normally.	Yes
SIGTERM (15)	Interrupt	Termination request. JVM will exit normally.	Yes
SIGHUP (1)	Interrupt	Hang up. JVM exits normally.	Yes
SIGQUIT	Control	A quit signal for a terminal. JVM uses this for taking Javadumps.	Yes
SIGTRAP (5)	Control	Used by the JIT.	Yes
__SIGRTMAX - 2	Control	Used by the SDK.	No

Table 23. Signals used by the JVM (continued)

Signal Name	Signal type	Description	Disabled by -Xrs
SIGCHLD	Control	Used by the SDK for internal control.	No

Use the **-Xrs** (reduce signal usage) option to prevent the JVM from handling most signals. For more information, see Sun's Java application launcher page.

Signals 1 (SIGHUP), 2 (SIGINT), 4 (SIGILL), 7 (SIGBUS), 8 (SIGFPE), 11 (SIGSEGV), and 15 (SIGTERM) on JVM threads cause the JVM to shut down; therefore, an application signal handler should not attempt to recover from these unless it no longer requires the JVM.

Linking a native code driver to the signal-chaining library

The Runtime Environment contains signal-chaining. Signal-chaining enables the JVM to interoperate more efficiently with native code that installs its own signal handlers.

Signal-chaining enables an application to link and load the shared library libjsig.so before the system libraries. The libjsig.so library ensures that calls such as signal(), sigset(), and sigaction() are intercepted so that their handlers do not replace the JVM's signal handlers. Instead, these calls save the new signal handlers, or "chain" them behind the handlers that are installed by the JVM. Later, when any of these signals are raised and found not to be targeted at the JVM, the preinstalled handlers are invoked.

To use libjsig.so:

- Link it with the application that creates or embeds a JVM:

  ```
  gcc -L$JAVA_HOME/bin -ljsig -L$JAVA_HOME/bin/j9vm -ljvm java_application.c
  ```

  or

- Use the **LD_PRELOAD** environment variable:

  ```
  export LD_PRELOAD=$JAVA_HOME/bin/libjsig.so; java_application (bash and ksh)

  setenv LD_PRELOAD=$JAVA_HOME/bin/libjsig.so; java_application (csh)
  ```

  (Assuming that `JAVA_HOME` is set up; otherwise, use /opt/ibm/java2-i386-50/jre.)

If you install signal handlers that use **sigaction()** , some **sa_flags** are not observed when the JVM uses the signal. These are:

- SA_NOCLDSTOP - This is always unset.
- SA_NOCLDWAIT - This is always unset.
- SA_RESTART - This is always set.

The libjsig.so library also hides JVM signal handlers from the application. Therefore, calls such as signal(), sigset(), and sigaction() that are made after the JVM has started no longer return a reference to the JVM's signal handler, but instead return any handler that was installed before JVM startup.

Working with floating stacks

Particular Linux distributions, for example Red Hat, have enabled a GLIBC feature called "floating stacks".

Because of Linux kernel limitations, the JVM does not run on SMP hardware with floating stacks enabled if the kernel level is less than 2.4.10. In this environment, floating stacks must be disabled before the JVM, or any application that starts the JVM, is started. On Red Hat, use this command to disable floating stacks by exporting an environment variable:

```
export LD_ASSUME_KERNEL=2.2.5
```

On a non-floating stack Linux system, regardless of what is set for **-Xss**, a minimum native stack size of 256 KB for each thread is provided. On a floating stack Linux system, the **-Xss** values are honored. Therefore, if you are migrating from a non-floating stack Linux system, you must ensure that any **-Xss** values are large enough and are not relying on a minimum of 256 KB.

Transforming XML documents

The IBM SDK contains the XSLT4J processor and the XML4J parser. These tools allow you to parse and transform XML documents independently from any given XML processing implementation. By using ″Factory Finders″ to locate the SAXParserFactory, DocumentBuilderFactory and TransformerFactory implementations, your application can swap between different implementations without having to change any code.

The IBM SDK contains the XSLT4J processor and the XML4J parser that conform to the JAXP 1.3 specification.

The XML technology included with the IBM SDK is similar to Apache Xerces Java and Apache Xalan Java. See http://xml.apache.org/xerces2-j/ and http://xml.apache.org/xalan-j/ for more information.

The XSLT4J processor allows you to choose between the original XSLT Interpretive processor or the new XSLT Compiling processor. The Interpretive processor is designed for tooling and debugging environments and supports the XSLT extension functions that are not supported by the XSLT Compiling processor. The XSLT Compiling processor is designed for high performance runtime environments; it generates a transformation engine, or *translet*, from an XSL style sheet. This approach separates the interpretation of style sheet instructions from their runtime application to XML data.

The XSLT Interpretive processor is the default processor. To select the XSLT Compiling processor, you can either:

- Change the entry in the jaxp.properties file (located in /opt/ibm/java2-i386-50/jre/lib).
- Set the system property for the **javax.xml.transform.TransformerFactory** key to org.apache.xalan.xsltc.trax.TransformerFactoryImpl.

To implement properties in the jaxp.properties file, copy jaxp.properties.sample to jaxp.properties in /opt/ibm/java2-i386-50/jre/lib. This file also contains full details about the procedure used to determine which implementations to use for the TransformerFactory, SAXParserFactory, and the DocumentBuilderFactory.

To improve the performance when you transform a StreamSource object with the XSLT Compiling processor, specify the com.ibm.xslt4j.b2b2dtm.XSLTCB2BDTMManager class as the provider of the service org.apache.xalan.xsltc.dom.XSLTCDTMManager. To determine the service provider, try each step until you find org.apache.xalan.xsltc.dom.XSLTCDTMManager:

1. Check the setting of the system property **org.apache.xalan.xsltc.dom.XSLTCDTMManager**.
2. Check the value of the property **org.apache.xalan.xsltc.dom.XSLTCDTMManager** in the file /opt/ibm/java2-i386-50/jre/lib/xalan.properties.
3. Check the contents of the file META-INF/services/org.apache.xalan.xsltc.dom.XSLTCDTMManager for a class name.
4. Use the default service provider, org.apache.xalan.xsltc.dom.XSLTCDTMManager.

The XSLT Compiling processor detects the service provider for the org.apache.xalan.xsltc.dom.XSLTCDTMManager service when a javax.xml.transform.TransformerFactory object is created. Any javax.xml.transform.Transformer or javax.xml.transform.sax.TransformerHandler objects that are created by using that TransformerFactory object will use the same service provider. You can change service providers by modifying one of the settings described above and then creating a new TransformerFactory object.

Using an older version of Xerces or Xalan

If you are using an older version of Xerces (prior to 2.0) or Xalan (prior to 2.3) in the endorsed override, you might get a null pointer exception when you launch your application. This exception occurs because these older versions do not handle the jaxp.properties file correctly.

If you are using an older version of Tomcat, this limitation might apply.

To avoid this situation, use one of the following workarounds:

- Upgrade to a newer version of the application that implements the latest Java API for XML Programming (JAXP) specification (http://java.sun.com/xml/jaxp/index.html).
- Remove the jaxp.properties file from /opt/ibm/java2-i386-50/jre/lib.
- Uncomment the entries in the jaxp.properties file in /opt/ibm/java2-i386-50/jre/lib.
- Set the system property for **javax.xml.parsers.SAXParserFactory**, **javax.xml.parsers.DocumentBuilderFactory**, or **javax.xml.transform.TransformerFactory** using the **-D** command-line option.
- Set the system property for **javax.xml.parsers.SAXParserFactory**, **javax.xml.parsers.DocumentBuilderFactory**, or **javax.xml.transform.TransformerFactory** from within your application's code. For an example, see the JAXP 1.3 specification.
- Explicitly set the SAX parser, Document builder, or Transformer factory using the **IBM_JAVA_OPTIONS** environment variable. For example:

```
export IBM_JAVA_OPTIONS=-Djavax.xml.parsers.SAXParserFactory=
  org.apache.xerces.jaxp.SAXParserFactoryImpl
```

or

```
export IBM_JAVA_OPTIONS=-Djavax.xml.parsers.DocumentBuilderFactory=
  org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
```

or

```
export IBM_JAVA_OPTIONS=-Djavax.xml.transform.TransformerFactory=
  org.apache.xalan.processor.TransformerFactoryImpl
```

Euro symbol support

The IBM SDK and Runtime Environment set the Euro as the default currency for those countries in the European Monetary Union (EMU) for dates on or after 1 January, 2002.

To use the old national currency, specify **–Duser.variant=PREEURO** on the Java command line.

If you are running the UK, Danish, or Swedish locales and want to use the Euro, specify **–Duser.variant=EURO** on the Java command line.

In V5.0 SR3, the default for the Slovenian locale is set to the Euro. If you install SR3 before 1 January 2007, you might want to change the currency to the Tolar.

Using the SDK to develop Java applications

The following sections give information about using the SDK for Linux to develop Java applications.

See "SDK tools" on page 196 for details of the tools available.

Debugging Java applications

To debug Java programs, you can use the Java Debugger (JDB) application or other debuggers that communicate by using the Java Platform Debugger Architecture (JPDA) that is provided by the SDK for Linux.

More information on problem diagnosis using Java can be found in the IBM Java Diagnostics Guide.

Java Debugger (JDB)

The Java Debugger (JDB) is included in the SDK for Linux. The debugger is invoked by the jdb command; it "attaches" to the JVM using JPDA.

To debug a Java application:

1. Start the JVM with the following options:

   ```
   java -Xdebug -Xrunjdwp:transport=dt_socket,server=y,address=<port>
        <class>
   ```

   The JVM starts up, but suspends execution before it starts the Java application.
2. In a separate session, you can attach the debugger to the JVM:

   ```
   jdb -attach <port>
   ```

   ```
   jdb -attach <port>
   ```

   The debugger will attach to the JVM, and you can now issue a range of commands to examine and control the Java application; for example, type `run` to allow the Java application to execute.

To find out more about JDB options, type:

```
jdb -help
```

To find out more about JDB commands:

1. Type `jdb`
2. At the jdb prompt, type `help`

You can also use JDB to debug Java applications running on remote machines. JPDA uses a TCP/IP socket to connect to the remote JVM.

1. Start the JVM with the following options:

   ```
   java -Xdebug -Xrunjdwp:transport=dt_socket,server=y,address=<port>
        <class>
   ```

   The JVM starts up, but suspends execution before it starts the Java application.
2. Attach the debugger to the remote machine:

   ```
   jdb -attach <host>:<port>
   ```

When you launch a debug session using the dt_socket transport, be sure that the specified ports are free to use.

The Java Virtual Machine Debugging Interface (JVMDI) is *not* supported in this release. It has been replaced by the Java Virtual Machine Tool Interface (JVMTI).

For more information on JDB and JPDA and their usage, see these Web sites:

- http://java.sun.com/products/jpda/
- http://java.sun.com/j2se/1.5.0/docs/guide/jpda/
- http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdb.html

Determining whether your application is running on a 32-bit or 64-bit JVM

Some Java applications must be able to determine whether they are running on a 32-bit JVM or on a 64-bit JVM. For example, if your application has a native code library, the library must be compiled separately in 32- and 64-bit forms for platforms that support both 32- and 64-bit modes of operation. In this case, your application must load the correct library at runtime, because it is not possible to mix 32- and 64-bit code.

The system property **com.ibm.vm.bitmode** allows applications to determine the mode in which your JVM is running. It returns the following values:

- 32 - the JVM is running in 32-bit mode
- 64 - the JVM is running in 64-bit mode

You can inspect the **com.ibm.vm.bitmode** property from within your application code using the call:

```
System.getProperty("com.ibm.vm.bitmode");
```

Writing JNI applications

Valid JNI version numbers that native programs can specify on the JNI_CreateJavaVM() API call are: JNI_VERSION_1_2(0x00010002) and JNI_VERSION_1_4(0x00010004).

Restriction: Version 1.1 of the Java Native Interface (JNI) is not supported.

This version number determines only the level of the JNI native interface to use. The actual level of the JVM that is created is specified by the JSE libraries (that is, V5.0). The JNI interface API *does not* affect the language specification that is implemented by the JVM, the class library APIs, or any other area of JVM behavior. For more information, see http://java.sun.com/j2se/1.5.0/docs/guide/jni.

If your application needs two JNI libraries, one built for 32- and the other for 64-bit, use the **com.ibm.vm.bitmode** system property to determine if you are running with a 31- or 64-bit JVM and choose the appropriate library.

To compile and link a native application with the SDK, use the following command:

```
gcc -I/opt/ibm/java2-i386-50/include -L/opt/ibm/java2-i386-50/jre/bin/j9vm
-ljvm -ldl -lpthread <JNI program filename>
```

The **-ljvm** option specifies that libjvm.so is the shared library that implements the JVM. The **-lpthread** option indicates that you are using native pthread support; if you do not link with the pthread library, a segmentation fault (signal SIGSEGV) might be caused when you run the JNI program.

Native formatting of Java types long, double, float

Previous versions of the SDK for z/OS 31-bit had a set of native conversion functions and macros for formatting large Java data types. These functions and macros were:

ll2str() function
Converts a jlong to an ASCII string representation of the 64-bit value.

flt2dbl() function
Converts a jfloat to a jdouble.

dbl2nat() macro
Converts a jdouble to an ESA/390 native double.

dbl_sqrt() macro
Calculates the square root of a jdouble and returns it as a jdouble.

dbl2str() function
Converts a jdouble to an ASCII string representation.

flt2str() function
Converts a jfloat to an ASCII string representation.

These functions and macros are not supported by Version 6 of the SDK for z/OS, and are no longer part of the libjvm.x interface. However, to provide a migration path, the functions have been moved to the demos area of the SDK and the appropriate demo code for these functions has been updated to reflect the changes.

The functions ll2str(), dbl2str(), and flt2str() are provided in the following object files:

- /opt/ibm/java2-i386-50/demo/jni/JNINativeTypes/c/convert.o (For 31-bit)
- /opt/ibm/java2-i386-50/demo/jni/JNINativeTypes/c/convert64.o (For 64-bit)

The function flt2dbl() and the macros dbl2nat() and dbl_sqrt() are not defined. However, the following macros give their definitions:

```
#include <math.h>
#define flt2dbl(f)  ((double)f)
#define dbl2nat(a)  ((a))
#define dbl_sqrt(a) (sqrt(a))
```

Note: These functions and macros are obsolete because the latest C/C++ compilers and runtimes can convert jlong, jdouble, and jfloat data types to strings by using printf()-type functions.

Support for thread-level recovery of blocked connectors

Four new IBM-specific SDK classes have been added to the com.ibm.jvm package to support the thread-level recovery of blocked connectors. The new classes are packaged in core.jar.

These classes allow you to unblock threads that have become blocked on networking or synchronization calls. If an application does not use these classes, it must end the whole process, rather than interrupting an individual blocked thread.

The classes are:

public Interface InterruptibleContext()
Defines two methods, isblocked() and unblock(). The other three classes implement InterruptibleContext.

public class InterruptibleLockContext()
A utility class for interrupting synchronization calls.

public class InterruptibleIOContext()
A utility class for interrupting network calls.

public class InterruptibleThread()
A utility class that extends java.lang.Thread, to allow wrapping of interruptible runnable methods. It uses instances of InterruptibleLockContext and InterruptibleIOContext to perform the required isblocked() and unblock() methods depending on whether a synchronization or networking operation is blocking the thread.

Both InterruptibleLockContext and InterruptibleIOContext work by referencing the current thread. Therefore if you do not use InterruptibleThread, you must provide your own class that extends java.lang.Thread, to use these new classes.

The Javadoc for these classes is provided with the SDK in the file docs/apidoc.zip.

Working with applets

With the Applet Viewer, you can run one or more applets that are called by reference in a Web page (HTML file) by using the APPLET tag. The Applet Viewer finds the APPLET tags in the HTML file and runs the applets, in separate windows, as specified by the tags.

Because the Applet Viewer is for viewing applets, it cannot display a whole Web page that contains many HTML tags. It parses only the APPLET tags and no other HTML on the Web page.

Running applets with the Applet Viewer

Use the following commands to run an applet with the Applet Viewer.

From a shell prompt, enter:

```
appletviewer <name>
```

where <*name*> is one of the following:

- The file name of an HTML file that calls an applet.
- The URL of a Web page that calls an applet.

For example, to invoke the Applet Viewer on an HTML file that calls an applet, type at a shell prompt:

```
appletviewer $HOME/<filename>.html
```

Where *filename* is the name of the HTML file.

For example, `http://java.sun.com/applets/NervousText/example1.html` is the URL of a Web page that calls an applet. To invoke the Applet Viewer on this Web page, type at a shell prompt:

```
appletviewer http://java.sun.com/applets/NervousText/example1.html
```

The Applet Viewer does not recognize the **charset** option of the <META> tag. If the file that the Applet Viewer loads is not encoded as the system default, an I/O exception might occur. To avoid the exception, use the **-encoding** option when you run appletviewer. For example:

```
appletviewer -encoding JISAutoDetect sample.html
```

Debugging applets with the Applet Viewer

You can debug applets by using the **-debug** option of the Applet Viewer. When debugging applets, you are advised to invoke the Applet Viewer from the directory that contains the HTML file that calls the applet.

For example:

```
cd demo/applets/TicTacToe
../../bin/appletviewer -debug example1.html
```

You can find documentation about how to debug applets using the Applet Viewer at the Sun Web site: http://java.sun.com.

Configuring large page memory allocation

You can enable large page support, on systems that support it, by starting Java with the **-Xlp** option.

Large page usage is primarily intended to provide performance improvements to applications that allocate a lot of memory and frequently access that memory. The large page performance improvements are mainly caused by the reduced number of misses in the Translation Lookaside Buffer (TLB). The TLB maps a larger virtual memory range and thus causes this improvement.

Large page support must be available in the kernel, and enabled, to allow Java to use large pages.

To configure large page memory allocation, first ensure that the running kernel supports large pages. Check that the file /proc/meminfo contains the following lines:

```
HugePages_Total:     <number of pages>
HugePages_Free:      <number of pages>
Hugepagesize:        <page size, in kB>
```

The number of pages available and their sizes vary between distributions.

If large page support is not available in your kernel, these lines will not exist in the /proc/meminfo file. In this case, you must install a new kernel containing support for large pages.

If large page support is available, but not enabled, `HugePages_Total` will be 0. In this case, your administrator must enable large page support. Check your operating system manual for more instructions.

For the JVM to use large pages, your system must have an adequate number of contiguous large pages available. If large pages cannot be allocated, even when enough pages are available, possibly the large pages are not contiguous.Configuring the number of large pages at bootup will create them contiguously.

Large page allocations will only succeed if the JVM has root access. To use large pages, either run Java as root or set the suid bit of the Java executable.

CORBA support

The Java 2 Platform, Standard Edition (J2SE) supports, at a minimum, the specifications that are defined in the compliance document from Sun. In some cases, the IBM J2SE ORB supports more recent versions of the specifications.

The minimum specifications supported are defined in the Official Specifications for CORBA support in J2SE (V1.5).

Support for GIOP 1.2

This SDK supports all versions of GIOP, as defined by chapters 13 and 15 of the CORBA 2.3.1 specification, OMG document .

http://www.omg.org/cgi-bin/doc?formal/99-10-07, *formal/99-10-07*.

Bidirectional GIOP is not supported.

Support for Portable Interceptors

This SDK supports Portable Interceptors, as defined by the OMG in the document *ptc/01–03–04*, which you can obtain from:

http://www.omg.org/cgi-bin/doc?ptc/01–03-04

Portable Interceptors are hooks into the ORB through which ORB services can intercept the normal flow of execution of the ORB.

Support for Interoperable Naming Service

This SDK supports the Interoperable Naming Service, as defined by the OMG in the document *ptc/00-08-07*, which you can obtain from:

http://www.omg.org/cgi-bin/doc?ptc/00-08-07

The default port that is used by the Transient Name Server (the tnameserv command), when no **ORBInitialPort** parameter is given, has changed from *900* to *2809*, which is the port number that is registered with the IANA (Internet Assigned Number Authority) for a CORBA Naming Service. Programs that depend on this default might have to be updated to work with this version.

The initial context that is returned from the Transient Name Server is now an org.omg.CosNaming.NamingContextExt. Existing programs that narrow the reference to a context org.omg.CosNaming.NamingContext still work, and do not need to be recompiled.

The ORB supports the **-ORBInitRef** and **-ORBDefaultInitRef** parameters that are defined by the Interoperable Naming Service specification, and the ORB::string_to_object operation now supports the ObjectURL string formats (corbaloc: and corbaname:) that are defined by the Interoperable Naming Service specification.

The OMG specifies a method ORB::register_initial_reference to register a service with the Interoperable Naming Service. However, this method is not available in the Sun Java Core API at Version 5.0. Programs that need to register a service in the current version must invoke this method on the IBM internal ORB implementation class. For example, to register a service "MyService":

```
((com.ibm.CORBA.iiop.ORB)orb).register_initial_reference("MyService",
serviceRef);
```

where `orb` is an instance of org.omg.CORBA.ORB, which is returned from ORB.init(), and `serviceRef` is a CORBA Object, which is connected to the ORB. This mechanism is an interim one, and is not compatible with future versions or portable to non-IBM ORBs.

System properties for tracing the ORB

A runtime debug feature provides improved serviceability. You might find it useful for problem diagnosis or it might be requested by IBM service personnel.

Tracing Properties

com.ibm.CORBA.Debug=*true*
Turns on ORB tracing.

com.ibm.CORBA.CommTrace=*true*
Adds GIOP messages (sent and received) to the trace.

com.ibm.CORBA.Debug.Output=<*filename*>
Specify the trace output file. By default, this is of the form orbtrc.DDMMYYYY.HHmm.SS.txt.

Example of ORB tracing

For example, to trace events and formatted GIOP messages from the command line, type:

```
java -Dcom.ibm.CORBA.Debug=true
     -Dcom.ibm.CORBA.CommTrace=true <myapp>
```

Limitations

Do not turn on tracing for normal operation, because it might cause performance degradation. Even if you have switched off tracing, FFDC (First Failure Data Capture) is still working, so serious errors are reported. If a debug output file is generated, examine it to check on the problem. For example, the server might have stopped without performing an ORB.shutdown().

The content and format of the trace output might vary from version to version.

System properties for tuning the ORB

The ORB can be tuned to work well with your specific network. The properties required to tune the ORB are described here.

Network tuning properties

com.ibm.CORBA.FragmentSize=<*size in bytes*>
Used to control GIOP 1.2 fragmentation. The default size is 1024 bytes.

To disable fragmentation, set the fragment size to 0 bytes:

```
java -Dcom.ibm.CORBA.FragmentSize=0 <myapp>
```

com.ibm.CORBA.RequestTimeout=<*time in seconds*>
Sets the maximum time to wait for a CORBA Request. By default the ORB waits indefinitely.

com.ibm.CORBA.LocateRequestTimeout=<*time in seconds*>
Set the maximum time to wait for a CORBA LocateRequest. By default the ORB waits indefinitely.

com.ibm.CORBA.ListenerPort=<*port number*>
Set the port for the ORB to read incoming requests on. If this property is set, the ORB starts listening as soon as it is initialized. Otherwise, it starts listening only when required.

Java 2 security permissions for the ORB

When running with a Java 2 SecurityManager, invocation of some methods in the CORBA API classes might cause permission checks to be made, which might result in a SecurityException. If your program uses any of these methods, ensure that it is granted the necessary permissions.

Security permissions required in the ORB

Table 24. Methods affected when running with Java 2 SecurityManager

Class/Interface	Method	Required permission
org.omg.CORBA.ORB	init	java.net.SocketPermission resolve
org.omg.CORBA.ORB	connect	java.net.SocketPermission listen
org.omg.CORBA.ORB	resolve_initial_references	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_is_a	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_non_existent	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	OutputStream _request (String, boolean)	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_get_interface_def	java.net.SocketPermission connect
org.omg.CORBA. Request	invoke	java.net.SocketPermission connect
org.omg.CORBA. Request	send_deferred	java.net.SocketPermission connect
org.omg.CORBA. Request	send_oneway	java.net.SocketPermission connect

Table 24. Methods affected when running with Java 2 SecurityManager (continued)

Class/Interface	Method	Required permission
javax.rmi. PortableRemoteObject	narrow	java.net.SocketPermission connect

ORB implementation classes

A list of the ORB implementation classes.

The ORB implementation classes in this release are:

- org.omg.CORBA.ORBClass=com.ibm.CORBA.iiop.ORB
- org.omg.CORBA.ORBSingletonClass=com.ibm.rmi.corba.ORBSingleton
- javax.rmi.CORBA.UtilClass=com.ibm.CORBA.iiop.UtilDelegateImpl
- javax.rmi.CORBA.StubClass=com.ibm.rmi.javax.rmi.CORBA.StubDelegateImpl
- javax.rmi.CORBA.PortableRemoteObjectClass=com.ibm.rmi.javax.rmi.PortableRemoteObject

These are the default values, and you are advised not to set these properties or refer to the implementation classes directly. For portability, make references only to the CORBA API classes, and not to the implementation. These values might be changed in future releases.

RMI over IIOP

Java Remote Method Invocation (RMI) provides a simple mechanism for distributed Java programming. RMI over IIOP (RMI-IIOP) uses the Common Object Request Broker Architecture (CORBA) standard Internet Inter-ORB Protocol (IIOP protocol) to extend the base Java RMI to perform communication. This allows direct interaction with any other CORBA Object Request Brokers (ORBs), whether they were implemented in Java or another programming language.

The following documentation is available:

- The RMI-IIOP Programmer's Guide is an introduction to writing RMI-IIOP programs. The guide is installed when you install this documentation.
- The demo directory contains:
  - A "Hello World" example that can switch between the Java Remote Method Protocol (JRMP) and IIOP protocols (demo/rmi-iiop/hello)
  - A "Hello World" example that interacts with a standard IDL program (demo/rmi-iiop/idl)
- The *Java Language to IDL Mapping* document is a detailed technical specification of RMI-IIOP and can be found at:http://www.omg.org/cgi-bin/doc?ptc/00-01-06.pdf

Implementing the Connection Handler Pool for RMI

Thread pooling for RMI Connection Handlers is not enabled by default.

To enable the connection pooling implemented at the RMI TCPTransport level, set the option

```
-Dsun.rmi.transport.tcp.connectionPool=true
```

This version of the Runtime Environment does not have a setting that you can use to limit the number of threads in the connection pool.

For more information, see the Sun Java site: http://java.sun.com.

Enhanced BigDecimal

From Java 5.0, the IBM BigDecimal class has been adopted by Sun as java.math.BigDecimal. The com.ibm.math.BigDecimal class is reserved for possible future use by IBM, and, currently, you are recommended to regard it as deprecated. You are recommended to migrate existing Java code to use java.math.BigDecimal.

The new java.math.BigDecimal uses the same methods as both the previous java.math.BigDecimal and com.ibm.math.BigDecimal. Existing code using java.math.BigDecimal continues to work correctly. The two classes do not serialize.

To migrate existing Java code to use the java.math.BigDecimal class, change the import statement at the top of your .java file from: `import com.ibm.math.*;` to `import java.math.*;`.

Support for XToolkit

The IBM SDK for Linux, v5.0 supports XToolkit. You need XToolkit when using the Eclipse's SWT_AWT bridge to build an application that uses both SWT and Swing. XToolkit is an alternative to the existing use of MToolkit libraries, with the benefit of faster rendering.

Related links:

- An example: Integrating Swing into Eclipse RCPs
- Reference Information in the Eclipse information center.
- Set up information on the Sun Web site.

Using the Java Communications API (JavaComm)

The Java Communications application programming interface (API) package (JavaComm) is an optional package provided for use with the Runtime Environment for Linux on the IA32, PPC32/PPC64, and AMD64/EM64T platforms. You install JavaComm independently of the SDK or Runtime Environment.

The JavaComm API gives Java applications a platform-independent way of performing serial and parallel port communications for technologies such as voice mail, fax, and smartcards. After writing serial or parallel port communications for your application, you can then include those files with your application.

The Java Communications API supports Electronic Industries Association (EIA)-232 (RS232) serial ports and Institute of Electrical and Electronics Engineers (IEEE) 1284 parallel ports and is supported on systems with the IBM Version 5.0 Runtime Environment.

Using Java Communications API, you can:

- List ports on a system
- Open and claim ownership of ports
- Resolve port ownership contention among applications that use Java Communications API
- Perform asynchronous and synchronous I/O port-monitoring through event notification
- Receive bean-style events describing state changes on the port

Installing Java Communications API

Make sure that a copy of the SDK or Runtime Environment is installed before you install the Java Communications API.

If you used the RPM package to install Java originally, install the Java Communications API from the RPM file. To install the Java Communications API from an RPM package, see "Installing the Java Communications API from an RPM file."

To install the Java Communications API from a .tgz file:

1. Put the Java Communications API zip file, ibm-java2-javacomm-50-linux-i386.tgz, in the directory where the SDK or Runtime Environment is installed. If you installed to the default directory, this is /opt/ibm/java2-i386-50/.
2. From a shell prompt, in the directory containing the .tgz file, extract the contents:

   ```
   tar -xvzf ibm-java2-javacomm-50-linux-i386.tgz
   ```

The Java Communications API is extracted into subdirectories within the existing directory.

Installing the Java Communications API from an RPM file

Make sure that a copy of the SDK or Runtime Environment is installed before you install the Java Communications API.

If you used the RPM package to install Java originally, install the Java Communications API from the RPM file.

1. Open a shell prompt, making sure you are root.
2. Use the `rpm -ivh` command to install the Java Communications API RPM file. For example:

   ```
   rpm -ivh ibm-java2-i386-javacomm-5.0-0.0.i386.rpm
   ```

The Java Communications API is installed within the /opt/ibm/java2-i386-50/ directory structure.

Location of Java Communications API files

Layout of the Java Communications API and the default installation directory.

By default, the Java Communications API files are installed in the /opt/ibm/java2-i386-50/ directory. The files and their structure are:

- jre/lib/ext/comm.jar
- jre/bin/libibmcomm.so
- jre/lib/javax.comm.properties

Configuring Java Communications API

After you install Java Communications API, you must:

- Change the access mode of serial and parallel ports.
- Set the PATH, if you did not set it when you installed Java. See "Setting the PATH" on page 199.

Changing the access mode of serial and parallel ports

After you install Java Communications API, you must change the access mode of serial and parallel ports so that users can access these devices.

You must give a user read/write access to the required devices. Log on as root and use the following commands, as applicable:

```
chmod 666 /dev/ttyS0    (also known as serial port COM1)
chmod 666 /dev/lp0      (also known as parallel port LPT1)
chmod 666 /dev/ttyS1    (also known as serial port COM2)
chmod 666 /dev/ttyS2    (also known as serial port COM3)
chmod 666 /dev/ttyS3    (also known as serial port COM4)
```

These commands give read/write access to everyone on the system.

An alternative method is to make the permissions 660 and add specific users to the group in which the devices reside. On a SUSE system, for example, the devices are in the `uucp` group. Thus, users can be added to the `uucp` group to gain access to the devices.

Change the access mode of any other ports as needed.

Specifying devices in the javax.comm.properties file:

The file javax.comm.properties allows you to specify the prefixes of the devices that are made available to the Java Communications API and whether they are parallel or serial. Port numbers are allocated sequentially to all devices.

For example, if you specify `/dev/ttyS=PORT_SERIAL` and the devices `/dev/ttyS0` and `/dev/ttyS1` exist, they will be allocated COM1 and COM2 respectively.

To use the USB-serial connectors, uncomment the line `/dev/ttyUSB=PORT_SERIAL` in the javax.comm.properties file. If the devices `/dev/ttyUSB0` and `/dev/ttyUSB1` exist and COM1 and COM2 have already been defined, the USB-serial devices are allocated the next sequential ports, COM3 and COM4.

Enabling serial ports on IBM ThinkPads

Most ThinkPads have their serial ports disabled by default in the BIOS. Currently, there is no way to enable the ports with Linux (the tpctl package *does not* enable the ports if they are disabled in the BIOS).

To enable the ports in the BIOS, you must use the DOS version of the ThinkPad Configuration Utility that is available from the IBM ThinkPad Download site. To use the ThinkPad Configuration Utility, you need a bootable DOS diskette. Note that the ThinkPad Configuration Utility might have been installed as part of the ThinkPad Utilities under Windows, depending on your installation options, and you can run it from a command prompt in Windows.

The ThinkPad Configuration application provided with Windows has options to enable or disable the serial and parallel ports but this *does not* also change the settings in the BIOS. So if you use this application with Windows, the ports are available; however, if you reboot your machine with Linux, the ports *will not* be enabled.

Printing limitation with the Java Communications API

When printing with the Java Communications API, you might have to select “Form feed”, “Continue”, or a similar option on the printer.

Uninstalling Java Communications API

The process you use to uninstall the Java Communications API depends on whether you installed the installable Red Hat Package Manager (RPM) package or the compressed Tape Archive (TAR) package.

Uninstalling the installable Red Hat Package Manager (RPM) package

Uninstalling the Java Communications API if you installed the installable RPM package.

1. `rpm -e ibm-java2-i386-javacomm-5.0-0.0` Alternatively, you can use a graphical tool such as kpackage or yast2
2. If the directory where you installed the Java Communications API does not contain any other tools that you require, remove that directory from your **PATH** statement.

Uninstalling the compressed Tape Archive (TAR) package

Uninstalling the Java Communications API, if you installed the compressed TAR package.

Delete the following files from the directory where you installed them:

- jre/lib/ext/comm.jar
- jre/bin/libibmcomm.so
- jre/lib/javax.comm.properties

Java Communications API documentation

You can find API documentation and samples for Java Communications API at the Sun Web site.

http://java.sun.com/products/javacomm/.

Deploying Java applications

This section discusses how to deploy applications written in Java.

Using the Java Plug-in

The Java Plug-in is a Web browser plug-in. If you use the Java plug-in, you can bypass your Web browser’s default JVM and instead use a Runtime Environment of your choice to run applets or beans in the browser.

You must allow applets to finish loading to prevent your browser from ’hanging’. For example, if you use the **Back** button and then the **Forward** button while an applet is loading, the HTML pages might be unable to load.

The Java plug-in is documented by Sun at: http://java.sun.com/j2se/1.5.0/docs/guide/plugin/developer_guide/.

Supported browsers

For Linux PPC32, Mozilla 1.6 is the supported browser.

For Linux IA32, see Table 25.

Table 25. Browsers supported by the Java Plug-in on Linux IA32

Distribution	Netscape default version	Netscape supported versions	Mozilla default version	Mozilla supported versions
Red Hat Enterprise Linux 3.0	-	7.x	1.4.2	1.4.1, 1.4.2, 1.7.12, Firefox 1.0.x, 1.5
Red Hat Enterprise Linux 4.0	4.8	7.x	1.7.3	1.4.1, 1.4.2, 1.7.12, Firefox 1.0.x, 1.5
SUSE Linux Enterprise Server 10	-	7.x	N/A	N/A
SUSE Linux Enterprise Server 9.0	-	7.x	1.6	1.4.1, 1.4.2, 1.6, 1.7.12, Firefox 1.0.x, 1.5

Installing and configuring the Java Plug-in

Multiple versions of the Java Plug-in are supplied with the SDK. You choose the right version for your browser. The most common are:

libjavaplugin_oji.so
The most common Plug-in. It is based on Mozilla's Open JVM Integration initiative and is used with most Mozilla products and derivatives, including Firefox.

libjavaplugin_ojigtk2.so
The same Plug-in as above but compiled with the GTK2 toolkit.

There are instructions for installing the Plug-in on some common browsers below.

You must symbolically link the Plug-in, rather than copy it, so that it can locate the JVM.

Mozilla:
Only Mozilla versions 1.4 and later are supported.

These steps will make the Java Plug-in available to all users

1. Log in as root.
2. Change to your Mozilla Plug-ins directory (this could be different on some Linux distributions).
   ```
   cd /usr/local/mozilla/plugins/
   ```
3. Create a symbolic link to libjavaplugin_oji.so.
   ```
   ln -s /opt/ibm/java2-i386-50/jre/bin/libjavaplugin_oji.so .
   ```

You must symbolically link the Plug-in, rather than copy it, so that it can locate the JVM.

Firefox:
These steps will make the Java Plug-in available to all users

1. Log in as root.

2. Change to your Firefox Plug-ins directory (this could be different on some Linux distributions).
   ```
   cd /usr/local/mozilla-firefox/plugins/
   ```
3. Create a symbolic link to libjavaplugin_oji.so.
   ```
   ln -s /opt/ibm/java2-i386-50/jre/bin/libjavaplugin_oji.so .
   ```

You must symbolically link the Plug-in, rather than copy it, so that it can locate the JVM.

Netscape 7.1 and above:
These steps will make the Java Plug-in available to all users

1. Log in as root.
2. Change to your Netscape Plug-ins directory (this could be different on some Linux distributions).
   ```
   cd /usr/local/netscape/plugins/
   ```
3. Create a symbolic link to javaplugin_oji.so.
   ```
   ln -s /opt/ibm/java2-i386-50/jre/bin/javaplugin_oji.so .
   ```

You must symbolically link the Plug-in, rather than copy it, so that it can locate the JVM.

Common Document Object Model (DOM) support

Because of limitations in particular browsers, you might not be able to implement all the functions of the org.w3c.dom.html package. One of the following errors will be thrown:

- A sun.plugin.dom.exception.NotSupportedException is thrown for some of these functions.
- A sun.plugin.dom.exception.InvalidStateException is thrown when the browser does not support a particular function.

Using DBCS parameters

The Java Plug-in supports double-byte characters (for example Chinese Traditional BIG-5, Korean, Japanese) as parameters for the tags `<APPLET>`, `<OBJECT>`, and `<EMBED>`. You must select the correct character encoding for your HTML document so that the Java Plug-in can parse the parameter. Specify character encoding for your HTML document by using the `<META>` tag in the `<HEAD>` section like this:

```
<meta http-equiv="Content-Type" content="text/html; charset=big5">
```

This example tells the browser to use the Chinese BIG-5 character encoding to parse the HTML file using. All the parameters are passed to the Java Plug-in correctly. However, some of the older versions of browsers might not understand this tag correctly. In this case, you can force the browser to ignore this tag, but you might have to change the encoding manually.

You can specify which encoding you want to use to parse the HTML file:

Netscape 4
View Menu → Character Set

Mozilla
View Menu → Character Coding

Using Web Start

Java Web Start is used to deploy Java applications.

Web Start allows you to launch and manage applications directly from the Web. Applications are cached to minimize installation times. Applications are automatically upgraded when new versions become available.

Web Start supports these java-vm-args documented at http://java.sun.com/j2se/1.5.0/docs/guide/javaws/developersguide/syntax.html#resources:

- -verbose
- -version
- -showversion
- -help
- -X
- -ea
- -enableassertions
- -da
- -disableassertions
- -esa
- -enablesystemassertions
- -dsa
- -disablesystemassertions
- -Xint
- -Xnoclassgc
- -Xdebug
- -Xfuture
- -Xrs
- -Xms
- -Xmx
- -Xss

IBM Web Start also supports **-Xgcpolicy** to set the garbage collection policy.

For information on the browsers that support Web Start, see Supported browsers.

For more information about Web Start, see: http://java.sun.com/products/javawebstart and http://java.sun.com/j2se/1.5.0/docs/guide/javaws/index.html. For more information about deploying applications, see: http://java.sun.com/j2se/1.5.0/docs/guide/deployment/index.html.

Running Web Start

Java Web Start Version 5.0 is installed automatically when you install Java using the .rpm or .tgz packages. If you extract Java from the .tgz package, run the jre/lib/javaws/updateSettings.sh shell script, to update the .mailcap and .mime.types files on your system.

You can invoke Web Start in a number of different ways.

- Select a link on a Web page that refers to a .jnlp file.
- At a shell prompt, type:

```
javaws <URL>
```

Where <*URL*> is the location of a .jnlp file.

- If you have used Java Web Start to open the application in the past, you can use the Java Application Cache Viewer. Run /opt/ibm/java2-i386-50/jre/bin/javaws.

  All Java Web Start applications are stored in the Java Application Cache. An application is only downloaded if the latest version is not in the cache.

Shipping Java applications

A Java application, unlike a Java applet, cannot rely on a Web browser for installation and runtime services.

When you ship a Java application, your software package probably consists of the following parts:

- Your own class, resource, and data files
- An installation procedure or program

To run your application, a user needs the Runtime Environment for Linux. The SDK for Linux software contains a Runtime Environment. However, you cannot assume that your users have the SDK for Linux software installed.

Your SDK for Linux software license does **not** allow you to redistribute any of the SDK's files with your application. You should ensure that a licensed version of the SDK for Linux is installed on the target machine.

Data sharing between JVMs for non Real-Time

Class data sharing is not supported when running with the **-Xrealtime** option.

The IBM Virtual Machine (JVM) allows you to share data between JVMs by storing it in a cache in shared memory. Sharing reduces the overall virtual memory consumption when more than one JVM shares a cache. Sharing also reduces the startup time for a JVM after the cache has been created. The shared class cache is independent of any active JVM and persists beyond the lifetime of the JVM that created the cache.

A shared cache can contain:

- Bootstrap classes
- Application classes
- Metadata that describes the classes

Overview of data sharing

The IBM SDK allows you to share classes between JVMs, while appearing transparent to the user.

Cache access

Any JVM may read or update the cache. The JVMs that are sharing must be at the same release.

You must take care if runtime bytecode modification is being used. See "Runtime bytecode modification" on page 239 for more information.

Dynamic updating of the cache

Because the shared class cache persists beyond the lifetime of any JVM, the cache is updated dynamically to reflect any modifications that might have been made to JARs or classes on the file system. The dynamic updating makes the cache transparent to the application using it.

Enabling class data sharing

Enable class data sharing by using the **-Xshareclasses** option when starting a JVM. The JVM will connect to an existing cache or create a new cache if one does not exist.

All bootstrap and application classes loaded by the JVM are shared by default. Custom class loaders share classes automatically if they extend the application class loader; otherwise, they must use the Java Helper API provided with the JVM to access the cache. (See "Adapting custom classloaders to share classes" on page 240.)

Cache security

Access to the shared class cache is limited by operating system permissions and Java security permissions. The shared class cache is created with user access by default unless the **groupAccess** command-line suboption is used. Only a class loader that has registered to share class data can update the shared class cache.

If a Java SecurityManager is installed, class loaders, excluding the default bootstrap, application, and extension class loaders, must be granted permission to share classes by adding SharedClassPermission lines to the java.policy file. (See "Using SharedClassPermission" on page 240.) The RuntimePermission "createClassLoader" restricts the creation of new class loaders and therefore also restricts access to the cache.

Cache lifespan

Multiple caches can exist on a system and are specified by name as a suboption to the **-Xshareclasses** command. A JVM can connect to only one cache at any one time.

You specify cache size on startup using **-Xscmx<n>[k|m|g]**, but this size is then fixed for the lifetime of the cache. Caches exist until they are explicitly destroyed using a suboption to the **-Xshareclasses** command or until the system is rebooted.

Cache utilities

All cache utilities are suboptions to the **-Xshareclasses** command.

See "Using command-line options for class data sharing," or use **-Xshareclasses:help** to see a list of available suboptions.

Using command-line options for class data sharing

Use the command-line options to enable and configure class data sharing.

For options that take a <*size*> parameter, you should suffix the number with ″k″ or ″K″ to indicate kilobytes, ″m″ or ″M″ to indicate megabytes, or ″g″ or ″G″ to indicate gigabytes.

-Xscmx<*size*>
: Specifies cache size. This option applies only if a cache is being created and no cache of the same name exists. Default cache size is platform-dependent. You can find out the size value being used by adding **-verbose:sizes** as a

command-line argument. Minimum cache size is 4 KB. Maximum cache size is also platform-dependent. (See “Cache size limits” on page 238.)

-Xshareclasses:<*suboption*>**[,**<*suboption*>**...]**
Enables class data sharing. Can take a number of suboptions, some of which are cache utilities. Cache utilities perform the required operation on the specified cache, without starting the VM. You can combine multiple suboptions, separated by commas, but the cache utilities are mutually exclusive. When running cache utilities, the message “Could not create the Java virtual machine” is expected. Cache utilities do not create the virtual machine.

You can use the following suboptions with the **-Xshareclasses** option:

help
Lists all the command-line options.

name=<*name*>
Connects to a cache of a given name, creating the cache if it does not already exist. Also used to indicate the cache that is to be modified by cache utilities, for example, destroy. Use the **listAllCaches** utility option to show which named caches are currently available. If you do not specify a name, the default name “sharedcc_%u” is used. %u in the cache name will insert the current user name. You can specify %g in the cache name to insert the current group name.

groupAccess
Sets operating system permissions on a new cache to allow group access to the cache. The default is user access only.

verbose
Enables verbose output, which gives feedback on cache activity.

verboseIO
Gives detailed output on the cache I/O activity, listing information on classes being stored and found. Each classloader is given a unique ID (the bootstrap loader is always 0) and the output shows the classloader hierarchy at work, where classloaders must ask their parents for a class before they can load it themselves. It is normal to see many failed requests; this behavior is expected for the classloader hierarchy.

verboseHelper
Enables verbose output for the Java Helper API. This output shows you how the Helper API is used by your ClassLoader.

silent
Turns off all shared classes messages, including error messages.

nonfatal
Allows the JVM to start even if class data sharing fails. Normal behavior for the VM is to refuse to start if class data sharing fails. If nonfatal is selected and the shared classes cache fails to initialize, the VM starts without class data sharing.

none
Can be added to the end of a command line to disable class sharing. This option overrides class sharing arguments found earlier on the command line.

modified=<*modified context*>
Used when a JVMTI agent is installed that might modify bytecode at runtime. If you do not specify this option and a bytecode modification agent is installed, classes will be safely shared with an extra performance

cost. The <*modified context*> is a descriptor chosen by the user; for example, "myModification1". This option partitions the cache, so that only VMs using context myModification1 can share the same classes. For instance, if you run HelloWorld with a modification context and then run it again with a different modification context, you will see all classes stored twice in the cache. See "Runtime bytecode modification" on page 239.

destroy (Utility option)
Destroys a cache using the name specified in the **name**=<*name*> suboption. If the name is not specified, the default cache is destroyed. A cache can be destroyed only if all VMs using it have shut down, and the user has sufficient permissions.

destroyAll (Utility option)
Tries to destroy all caches available to the user. A cache can be destroyed only if all VMs using it have shut down, and the user has sufficient permissions.

expire=<*time in minutes*>
Destroys all caches that have been unused for the time specified before loading shared classes.

listAllCaches (Utility option)
Lists all the caches on the system, describing if they are in use and when they were last used.

printStats (Utility option)
Displays summary statistics information about the cache specified in the **name**=<*name*> suboption. If the name is not specified, statistics are displayed about the default cache. The most useful information displayed is how full the cache is and how many classes it contains. Stale classes are classes that have been updated on the file system and which the cache has therefore marked ″stale″. Stale classes are not purged from the cache and can be reused. See the Diagnostics Guide for more information.

printAllStats (Utility option)
Displays detailed information about the contents of the cache specified in the **name**=<*name*> suboption. If the name is not specified, statistics are displayed about the default cache. Every class is listed in chronological order along with a reference to the location from which it was loaded. See the Diagnostics Guide for more information.

Creating, populating, monitoring, and deleting a cache

To enable class data sharing, add **-Xshareclasses[:name=<name>]** to your application command line.

The JVM will either connect to an existing cache of the given name, or create a new cache of that name. If a new cache has been created, it will be populated with all bootstrap and application classes being loaded until the cache becomes full. If two or more JVMs are started concurrently, they will all populate the cache concurrently.

To check that the cache has been created, run `java -Xshareclasses:listAllCaches`. To see how many classes and how much class data is being shared, run `java -Xshareclasses:[name=<name>],printStats`. (These utilities can be run after the application JVM has terminated or in another command window.)

For more feedback on cache usage while the JVM is running, use the **verbose** suboption. For example, `java -Xshareclasses:[name=<name>],verbose`.

To see classes being loaded from the cache or stored in the cache, add `-Xshareclasses:[name=<name>],verboseIO` to your application command line.

To delete the cache created, run `java -Xshareclasses:[name=<name>],destroy`. You should have to delete caches only if they contain many stale classes or if the cache is full and you want to create a bigger cache.

You are recommend to tune the cache size for your specific application, because the default is unlikely to be the optimum size. The best way to determine the optimum cache size is to specify a large cache (using **-Xscmx**), run the application, and then use printStats to determine how much class data has been stored. Add a small amount to the value shown in printStats for contingency. Note that because classes can be loaded at any time during the lifetime of the JVM, it is best to do this analysis after the application has terminated. However, a full cache does not have a negative impact on the performance or capability of any JVMs connected to it, so it is quite legitimate to decide on a cache size that is smaller than required.

If a cache becomes full, a message is output to the command line of any JVMs using verbose suboption. All JVMs sharing the full cache will then load any further classes into their own process memory. Classes in a full cache can still be shared, but a full cache is read-only and cannot be updated with new classes.

Performance and memory consumption

Class data sharing is particularly useful on systems that use more than one JVM running similar code; the system benefits from reduced virtual memory consumption. It is also useful on systems that frequently start up and shut down JVMs, which benefit from the improvement in startup time.

The overhead to create and populate a new cache is minimal. The JVM startup cost in time for a single JVM is typically between 0 and 5% slower compared with a system not using class data sharing, depending on how many classes are loaded. JVM startup time improvement with a populated cache is typically between 10% and 40% faster compared with a system not using class data sharing, depending on the operating system and the number of classes loaded. Multiple JVMs running concurrently will show greater overall startup time benefits.

Duplicate classes are consolidated within the shared class cache. For example, class A loaded from myClasses.jar and class A loaded from myOtherClasses.jar (with identical content) is stored only once in the cache. The **printAllStats** utility shows multiple entries for duplicated classes, with each entry pointing to the same class.

When you run your application with class data sharing, you can use the operating system tools to see the reduction in virtual memory consumption.

Limitations and considerations of using class sharing

Some factors that you should consider when using class sharing.

Cache size limits

The cache for sharing classes is allocated using System V IPC Shared memory mechanism.

The maximum theoretical cache size is 2 GB. The size of cache you can specify is limited by the amount of physical memory and paging space available to the system.

Because the virtual address space of a process is shared between the shared classes cache and the Java heap, increasing the maximum size of the Java heap will reduce the size of the shared classes cache you can create.

Cache size is limited by **SHMMAX** settings, which limits the amount of shared memory that can be allocated. You can find these settings by looking at /proc/sys/kernel/shmmax file. **SHMMAX** is usually set to 30MB.

Runtime bytecode modification

Any JVM using a JVMTI (JVM Tool Interface) agent that can modify bytecode data should use the modified=<*modified_context*> suboption if it wants to share the modified classes with another VM. (See "Using command-line options for class data sharing" on page 235.) The modified context is a user-specified descriptor that describes the type of modification being performed. The modified context partitions the cache so that all JVMs running under the same context share a partition. This partitioning allows JVMs that are not using modified bytecode to safely share a cache with those that are using modified bytecode. All JVMs using a given modified context must modify bytecode in a predictable, repeatable manner for each class, so that the modified classes stored in the cache have the expected modifications when they are loaded by another VM. Any modification must be predictable because classes loaded from the shared class cache cannot be modified again by the agent.

If a JVMTI agent is used without a modification context, classes are still safely shared by the VM, but with a small impact on performance. Using a modification context with a JVMTI agent avoids the need for extra checks and therefore has no impact on performance. A custom ClassLoader that extends java.net.URLClassLoader and modifies bytecode at load time without using JVMTI automatically stores that modified bytecode in the cache, but the cache does not treat the bytecode as modified. Any other VM sharing that cache loads the modified classes. The modified=<*modification_context*> suboption can be used in the same way as with JVMTI agents to partition modified bytecode in the cache. If a custom ClassLoader needs to make unpredictable load-time modifications to classes, that ClassLoader must not attempt to use class data sharing.

See the Diagnostics Guide for more detail on this topic.

Operating system limitations

For operating systems that can run both 32-bit and 64-bit applications, class data sharing is not permitted between 32-bit and 64-bit JVMs. The **listAllCaches** suboption lists 32-bit or 64-bit caches, depending on the address mode of the JVM being used.

The shared class cache requires disk space to store identification information about the caches that exist on the system. This information is stored in /tmp/javasharedresources. If the identification information directory is deleted, the JVM cannot identify the shared classes on the system and must recreate the cache. Use the `ipcs` command to view the memory segments used by a JVM or application.

Users running a JVM must be in the same group to use a shared class cache. The permissions for accessing a shared classes cache are enforced by the operating

system. If a cache name is not specified, the user name is appended to the default name so that multiple users on the same system create their own caches by default.

Using SharedClassPermission

If a SecurityManager is being used with class data sharing and the running application uses its own class loaders, these class loaders must be granted shared class permissions before they can share classes. You add shared class permissions to the java.policy file using the ClassLoader class name (wildcards are permitted) and either “read”, “write”, or “read,write” to determine the access granted. For example:

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

If a ClassLoader does not have the correct permissions, it is prevented from sharing classes. You cannot change or reduce the permissions of the default bootstrap, application, or extension class loaders.

Adapting custom classloaders to share classes

Most Java applications use the VM’s own class loaders or have custom class loaders that extend java.net.URLClassLoader.

Applications using these class loaders can automatically share bootstrap and application classes. Custom classloaders that do not extend java.net.URLClassLoader will require modifications to make use of class sharing. All custom class loaders must be granted shared class permissions if a SecurityManager is being used, see “Using SharedClassPermission.” IBM provides several Java interfaces for various types of custom class loaders, which allow the class loaders to find and store classes in the shared class cache. These classes are in the com.ibm.oti.shared package. The Javadoc for this package is provided with the SDK in the file docs/apidoc.zip. See the Diagnostics Guide for more information on how to use these interfaces.

Service and support for independent software vendors

Contact points for service:

If you are entitled to services for the Program code pursuant to the IBM Solutions Developer Program, contact the IBM Solutions Developer Program through your normal method of access or on the Web at: http://www.ibm.com/partnerworld/.

If you have purchased a service contract (that is, IBM’s Personal Systems Support Line or equivalent service by country), the terms and conditions of that service contract determine what services, if any, you are entitled to receive with respect to the Program.

Accessibility

The User Guides that are supplied with this SDK and the Runtime Environment have been tested by using screen readers. You can use a screen reader such as the IBM Home Page Reader or the JAWS screen reader with these User Guides.

To change the font sizes in the User Guides, use the function that is supplied with your browser, usually found under the **View** menu option.

For users who require keyboard navigation, a description of useful keystrokes for Swing applications is in *Swing Key Bindings* at http://www.ibm.com/developerworks/java/jdk/additional/.

iKeyman command line tool

In addition to the GUI, the iKeyman tool provides the command-line tool, ikeycmd, which has the same functions that the iKeyman GUI has. ikeycmd allows you to manage keys, certificates, and certificate requests. You can call ikeycmd from native shell scripts and from programs that are to be used when applications need to add custom interfaces to certificate and key management tasks.

ikeycmd can create key database files for all the types that iKeyman currently supports. ikeycmd can also create certificate requests, import CA signed certificates, and manage self-signed certificates.

Usage

To run an ikeycmd command, enter:

java [**-Dikeycmd.properties=**<*properties file*>] **com.ibm.gsk.ikeyman.ikeycmd** <*object*> <*action*> [*options*]

Important: The <*object*> parameter **must** come before the <*action*> parameter.

The parameters used are:

<*object*>
The object to act upon is specified as one of the following types:

-keydb
Specifies a key database (either a CMS key database file, a WebDB keyring file, or SSLight class)

-cert
Specifies a certificate within a key database.

-certreq
Specifies a certificate request within a key database.

<*action*>
The specific action that is to be taken on the object. To see the available actions for an object, invoke ikeycmd passing only the object as an argument. Context-sensitive help shows the available actions for that object.

For example:

```
$ java com.ibm.gsk.ikeyman.ikeycmd -keydb
A key database action must be specified for this command.
Object   Action       Description
----     ------       -----------------------------------------------------------
-keydb   -changepw    Change the password for a key database
         -convert     Convert the format of a key database
         -create      Create a key database
         -delete      Delete a key database
         -expiry      Display password expiry
         -stashpw     Stash the password of a key database into a file
         -list        Currently supported types of key database.
```

[options]
Different options are available depending on the action being taken. To see the

available options for an action, invoke ikeycmd passing the object, the action and the **-help** option. Context-sensitive help shows the available actions for that object.

For example:

```
$ java com.ibm.gsk.ikeyman.ikeycmd -keydb -create -help
-help is not a valid option. Proper syntax is:

-keydb -create -db <name> [-pw <passwd>] [-type <cms | jks | jceks | pkcs12>]
               [-expire <days>] [-stash] [-usereasoncode]
```

-Dikeycmd.properties
Specifies the name of an optional properties file to use for this Java invocation. A default properties file, ikeycmd.properties, is provided as a sample file that can be changed and used by any Java application.

-version
Displays version information.

-help
Displays usage information.

Further Reading

For more information, see the iKeyman User Guide at: http://www.ibm.com/developerworks/java/jdk/security/index.html.

Keyboard traversal of JComboBox components in Swing

If you traverse the drop-down list of a JComboBox component with the cursor keys, the button or editable field of the combo box does not change value until an item is selected. This is the desired behavior for this release and improves accessibility and usability by ensuring that the keyboard traversal behavior is consistent with mouse traversal behavior.

Web Start accessibility (Linux IA 32-bit, PPC32, and PPC64 only)

From Version 5.0, Java Web Start contains several accessibility and usability improvements, including better support for screen readers and improved keyboard navigation.

You can use the command line only to launch a Java application that is enabled for Web Start. To change preference options, you must edit a configuration file, .java/.deployment/.deployment.properties in the user's home directory. Take a backup before you edit this file. Not all the preferences that can be set in the Java Application Cache Viewer are available in the configuration file.

Known limitations

Known limitations on the SDK and Runtime Environment for Linux.

You can find more help with problem diagnosis in the *IBM Diagnostics Guide* at http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html.

JConsole monitoring tool Local tab

In IBM's JConsole tool, the **Local** tab, which allows you to connect to other Virtual Machines on the same system, is not available. Also, the corresponding

command-line **pid** option is not supported. Instead, use the **Remote** tab in JConsole to connect to the Virtual Machine that you want to monitor. Alternatively, use the **connection** command-line option, specifying a host of `localhost` and a port number. When you launch the application that you want to monitor, set these command-line options:

-Dcom.sun.management.jmxremote.port=<*value*>
Specifies the port the management agent should listen on.

-Dcom.sun.management.jmxremote.authenticate=false
Disables authentication unless you have created a username file.

-Dcom.sun.management.jmxremote.ssl=false
Disables SSL encryption.

Slow DSA key pair generation

Creating DSA key pairs of unusual lengths can take a significant amount of time on slow machines. Do not interpret the delay as a hang because the process will complete if sufficient time is allowed. The DSA key generation algorithm has been optimized to generate standard key lengths (for instance, 512, 1024) more quickly than others.

Window managers and keyboard shortcuts

Your window manager might override some of the Java keyboard shortcuts. If you need to use an overridden Java keyboard shortcut, consult your operating system manual and change your window manager keyboard shortcuts.

X Windows file descriptors

The X Windows System is unable to use file descriptors above 255. Because the JVM holds file descriptors for open jar files, X can run out of file descriptors. As a workaround, you can set the **JAVA_HIGH_ZIPFDS** environment variable to tell the JVM to use higher file descriptors for jar files.

To use the **JAVA_HIGH_ZIPFDS** environment variable, set it to a value between 0 and 512. The JVM will then open the first jar files using file descriptors up to 1024. For example, if your program is likely to load 300 jar files:

```
export JAVA_HIGH_ZIPFDS=300
```

The first 300 jar files will then be loaded using the file descriptors 724 to 1023. Any jar files opened after that will be opened in the normal range.

DBCS and the KDE clipboard

You might not be able to use the system clipboard with double-byte character set (DBCS) to copy information between Linux applications and Java applications if you are running the K Desktop Environment (KDE).

ThreadMXBean Thread User CPU Time limitation

In package java.lang.management, the methods ThreadMXBean.getThreadUserTime() and ThreadMXBean.getCurrentThreadUserTime() are not supported. These methods always return -1. These methods are not supported even when

ThreadMXBean.isThreadCpuTimeSupported() and ThreadMXBean.isCurrentThreadCpuTimeSupported() return true.

This limitation does not affect ThreadMXBean.getThreadCpuTime() or ThreadMXBean.getCurrentThreadCpuTime().

Stopping Java applications with Ctrl-C

If you stop a Java application with **Ctrl-C**, several processes might be left orphaned.

KeyEvents and window managers

KeyEvent results that include the **Alt** key might differ between window managers in Linux. They also differ from results of other operating systems. In the Enlightenment window manager, **Alt+A** and **Shift+Alt+A** produce different modifiers, and **Ctrl+Alt+A** produces no key event at all. However, using the WindowMaker window manager, **Ctrl+Alt+A** produces a key event.

X Windows and the Meta key

On the Linux X Window System, the keymap is set to: 64 0xffe9 (Alt_L) 0xffe7 (Meta_L), and 113 0xffea (Alt_R) 0xffe8 (Meta_R). You can check this by typing the following at a shell prompt:

```
xmodmap -pk
```

This is why the SDK considers that Meta and Alt are being pressed together. As a workaround, you can remove the Meta_x mapping by typing the following at a shell prompt:

```
xmodmap -e "keysym Alt_L = Alt_L" -e "keysym Alt_R = Alt_R"
```

This workaround might affect other X-Window applications that are running on the same display if they use the Meta-key that was removed.

Creating a JVM using JNI

Native programs cannot create a VM with JNI_VERSION_1_1(0x00010001) interfaces. You cannot call JNI_CreateJavaVM() and pass it a version of JNI_VERSION_1_1(0x00010001). The versions that can be passed are:

- JNI_VERSION_1_2(0x00010002)
- JNI_VERSION_1_4(0x00010004)

The VM created is determined by the Java libraries present (that is, 1.2.2, 1.3.x, 1.4.x, 5.x), not the one that is implied by the JNI interface version passed.

The interface version does not affect any area of VM behavior other than the functions available to native code.

SIGSEGV when creating a JVM using JNI

A call to JNI_CreateJavaVM() from a JNI application might cause a segmentation fault (signal SIGSEGV); to avoid this fault, rebuild your JNI program specifying the option **–lpthread**.

Traditional Chinese input and the Enter, Space, Backspace, and Esc keys

For users of Traditional Chinese only, function keys **Enter**, **Space**, **Backspace**, and **Esc** do not work as expected when using the LinYi ZhuYin, YiTian ZhuYin, and Bosham IMEs to input the TCH characters. The **Enter** key inserts a new line, the **Esc** key is used to cancel the preedit strings or the candidate window, and the **Space** key inserts a space. The candidate window should disappear after the commit key has been pressed and it should not appear when you press the **Backspace** key while using the LinYi ZhuYin, YiTian ZhuYin, and Bosham IMEs to input the Traditional Chinese characters.

Lack of resources with highly-threaded applications

If you are running with a lot of concurrent threads, you might get a warning message:

```
java.lang.OutOfMemoryError
```

This is an indication that your machine is running out of system resources and messages can be caused by the following reasons:

- The number of processes created exceeds your user limit. (If your Linux installation uses LinuxThreads, rather than NPTL.)
- Not enough system resources are available to create new threads. In this case, you might see also other Java exceptions, depending on what your application is running.
- Kernel memory is either running out or is fragmented. You can see corresponding Out of Memory kernel messages in /var/log/messages that have the killed process id.

Try tuning your system to increase the corresponding system resources.

Upgrading an SDK distributed by Red Hat

During installation, if you have already installed an earlier version of this product that was distributed by Red Hat, the Red Hat Package Manager (RPM) will report that the IBM package is older than any versions of the product that Red Hat has distributed. This condition occurs because Red Hat adds to the RPM package a "serial number" that overrides the versioning scheme of the product in the package, and, by default, the RPM refuses to install a package that seems to be older than one already installed. The **--oldpackage** option of the rpm command enables you to install the new IBM package.

X Server and client font problems

When running a Java AWT or Swing application on a Linux machine and exporting the display to a second machine, you might experience problems displaying some dialogs if the set of fonts loaded on the X client machine is different from the set loaded on the X server machine. This problem is noticeable particularly with SLES8 in Chinese locales, where the default installation of the server version installs arphic fonts, but the default installation of the client does not. To avoid this problem, install the same fonts on both machines.

UTF-8 encoding and MalformInputExceptions

If your system locale is using a UTF-8 encoding, some SDK tools might throw a sun.io.MalformedInputException. To find out whether your system is using a UTF-8 encoding, examine the locale-specific environment variables such as **LANG** or **LC_ALL** to see if they end with the suffix ".UTF-8". If you get this sun.io.MalformedInputException, change characters that are not within the 7-bit ASCII range (0x00 - 0x7f) and are not represented as Java Unicode character literals to Java Unicode character literals (for example: '\u0080'). You can also work around this problem by removing the ".UTF-8" suffix from the locale-specific environment variables; for example, if your machine has default locale of "en_US.UTF-8", set **LANG** to "en_US".

AMI and xcin problems when exporting displays

If you are using AMI and xcin in a cross-platform environment (for example, if you try to export the display between a 32-bit and a 64-bit system, or between a big-endian and a little-endian system) there might be some problems. If you have this kind of problem, contact your Linux distributor.

RHEL3, AWT and Chinese characters

On Red Hat Enterprise Linux 3, Chinese characters might not be displayed in AWT text components except for GB2312 characters in the Chinese GB18030 locale.

RHEL4 and XIM

For Chinese, Korean and Japanese language users of Red Hat Enterprise Linux 4 only. No XIM server is installed by default. To input DBCS characters to a Java application, please install a XIM server package such as iiimf-x or kinput2.

RHEL4 and IIIMF

For Chinese, Korean, and Japanese language users of Red Hat Enterprise Linux 4 only, if you are using the Internet/Intranet Input Method Framework (IIIMF), use IIIMF packages that are included in Red Hat Enterprise Linux 4 Update 2 or later. Also, note the following problem. If you experience this problem, contact Red Hat for guidance, at http://www.redhat.com.

On Linux zSeries 64-bit, you might experience IIIMF failures or a failure to start. See Red Hat Bugzilla number RIT73533 for Japanese users only.

RHEL4 and the zh_CN.GB18030 locale

For Simplified Chinese language users of Red Hat Enterprise Linux 4 only. The zh_CN.GB18030 locale is not supported by xlib in RHEL4. xterm cannot activate Input Method Server to input GB18030 characters. Use the zh_CN.UTF8 locale instead. If you have legacy programs or data encoded with GB2312, GBK, or GB18030, and you want to migrate them to RHEL4, you must preprocess them with iconv to convert them to UTF-8 encoding so that the programs can run and data can be displayed properly in RHEL4 with the zh_CN.UTF8 locale.

This limitation is resolved in RHEL4 U3.

RHEL4 and xcin

64-bit environments only

On RHEL4 with xcin (Traditional Chinese XIM server), you might experience unexpected behavior such as a segementation fault with Java on 64-bit environments (such as AMD64 or zSeries 64-bit platforms). Contact Red Hat for guidance, at http://www.redhat.com. See Red Hat Bugzilla number RIT78430.

RHEL4 and IIIMF focus change problems

For Red Hat Enterprise Linux 4, when using IIIMF (Internet Intranet Input Method Framework) to input DBCS characters, you may encounter focus change problems. The problem occurs when minimizing active input components. After restoring the component, the input method will switch back to SBCS. DBCS must then be manually reactivated.

The following components have this focus change problem:

- java.awt.Canvas
- java.awt.Button
- javax.swing.JButton
- javax.swing.JSplitPane
- javax.swing.JComboBox
- javax.swing.JList

XIM and the Java Plug-in

For Japanese, Chinese, and Korean language users, you cannot use XIM to input your own characters into text components on a Java applet in a Web browser. This limitation occurs because XEmbed requires a fix to the X11 library file. To work around this situation, specify the **-Dsun.awt.noxembed=true** system parameter to disable XEmbed. You can set this option by using the control panel:

1. Open the Java Plug-in control panel and go to the **Java** tab.
2. Click the **View** button in the Java Applet Runtime Settings.
3. Input **-Dsun.awt.noxembed=true** in the Java Runtime Parameters and click **OK**.
4. Click **Apply**.
5. Start a browser.

This limitation is resolved in RHEL4 U3.

Arabic characters and Matrox video cards

Intel 32-bit platforms only

For Arabic text users, when using Linux with a Matrox video card and acceleration enabled, distortion of characters can be seen when using drawString to display large fonts. This problem is caused by the driver for those cards. The suggested workaround is to disable acceleration for the device.

SLES9 NPTL and the parallel port driver

Intel 32-bit platforms only

On SLES 9 NPTL, the parallel port driver causes a kernel crash and brings down a Java thread. The JVM detects this crash when it tries to suspend the thread for Garbage Collection and then crashes producing a core file and the message "JVMLH030: threads are disappearing when trying to suspend all threads".

SUSE Bugzilla report 47947 is raised against this problem. This bug is fixed in SLES 9 Service Pack 1.

Appendix B. RMI-IIOP Programmer's Guide

Discusses how to write Java Remote Method Invocation (RMI) programs that can access remote objects by using the Internet Inter-ORB Protocol (IIOP).

Preface

This document discusses how to write Java Remote Method Invocation (RMI) programs that can access remote objects by using the Internet Inter-ORB Protocol (IIOP). By making your RMI programs conform to a small set of restrictions (see “Restrictions when running RMI programs over IIOP” on page 255), your RMI programs can access CORBA objects. RMI-IIOP gives you RMI ease-of-use coupled with CORBA/IIOP language interoperability.

What are RMI, IIOP, and RMI-IIOP?

The basic concepts behind RMI-IIOP and other similar technologies.

RMI

With RMI, you can write distributed programs in the Java programming language. RMI is easy to use, you do not need to learn a separate interface definition language (IDL), and you get Java's inherent ″write once, run anywhere″ benefit. Clients, remote interfaces, and servers are written entirely in Java. RMI uses the Java Remote Method Protocol (JRMP) for remote Java object communication. For a quick introduction to writing RMI programs, see the RMI tutorial web page, which describes writing a simple ″Hello World″ RMI program.

RMI lacks interoperability with other languages, and, because it uses a non-standard communication protocol, cannot communicate with CORBA objects.

IIOP, CORBA, and Java IDL

IIOP is CORBA's communication protocol. It defines the way bits are sent over a wire between CORBA clients and servers. CORBA is a standard distributed object architecture developed by the Object Management Group (OMG). Interfaces to remote objects are described in a platform-neutral interface definition language (IDL). Mappings from IDL to specific programming languages are implemented, binding the language to CORBA/IIOP.

The Java 2 Platform, Standard Edition (J2SE) V5.0 CORBA/IIOP implementation is known as Java IDL. Along with the IDL to Java (idlj) compiler, Java IDL can be used to define, implement, and access CORBA objects from the Java programming language.

The Java IDL web page gives you a good, Java-centric view of CORBA/IIOP programming. To get a quick introduction to writing Java IDL programs, see the Getting Started: Hello World web page.

RMI-IIOP

Previously, Java programmers had to choose between RMI and CORBA/IIOP (Java IDL) for distributed programming solutions. Now, by adhering to a few restrictions (see “Restrictions when running RMI programs over IIOP” on page 255), RMI server objects can use the IIOP protocol, and communicate with CORBA client objects written in any language. This solution is known as RMI-IIOP. RMI-IIOP combines RMI ease of use with CORBA cross-language interoperability.

Using RMI-IIOP

This section describes how to use the IBM RMI-IIOP implementation.

The rmic compiler

Reference information on the rmic compiler.

Purpose

The rmic compiler generates IIOP stubs and ties and emits IDL, in accordance with the Java Language to OMG IDL Language Mapping Specification.

Parameters

-iiop

Generates stub and tie classes. A stub class is a local proxy for a remote object. Clients use stub classes to send calls to a server. Each remote interface requires a stub class, which implements that remote interface. The client's reference to a remote object is a reference to a stub. Tie classes are used on the server side to process incoming calls, and dispatch the calls to the proper implementation class. Each implementation class requires a tie class.

Stub classes are also generated for abstract interfaces. An abstract interface is an interface that does not extend java.rmi.Remote, but has methods that throw either java.rmi.RemoteException or a superclass of java.rmi.RemoteException. Interfaces that do not extend java.rmi.Remote and have no methods are also abstract interfaces.

Using the **-poa** option changes the inheritance from org.omg.CORBA_2_3.portable.ObjectImpl to org.omg.PortableServer.Servant. This type of mapping is nonstandard and is not specified by the Java Language to OMG IDL Mapping Specification.

The PortableServer module for the Portable Object Adapter (POA) defines the native Servant type. In the Java programming language, the Servant type is mapped to the Java org.omg.PortableServer.Servant class. It serves as the base class for all POA servant implementations and provides a number of methods that may be invoked by the application programmer, as well as methods that are invoked by the POA itself and may be overridden by the user to control aspects of servant behavior.

-poa

Changes the inheritance from org.omg.CORBA_2_3.portable.ObjectImpl to org.omg.PortableServer.Servant. This type of mapping is nonstandard and is not specified by the Java Language to OMG IDL Mapping Specification.

The PortableServer module for the Portable Object Adapter (POA) defines the native Servant type. In the Java programming language, the Servant type is mapped to the Java org.omg.PortableServer.Servant class. It serves as the base class for all POA servant implementations and provides a number of methods that may be invoked by the application programmer, as well as methods that are invoked by the POA itself and may be overridden by the user to control aspects of servant behavior.

Valid only when the **-iiop** option is present.

-idl

Generates OMG IDL for the classes specified and any classes referenced. This option is required only if you have a CORBA client written in another language that needs to talk to a Java RMI-IIOP server.

Tip: After the OMG IDL is generated using **rmic -idl**, use the generated IDL with an IDL-to-C++ or other language compiler, but not with the IDL-to-Java language compiler. ″Round tripping″ is not recommended and should not be necessary. The IDL generation facility is intended to be used with other languages. Java clients or servers can use the original RMI-IIOP types.

IDL provides a purely declarative, programming-language-independent means of specifying the API for an object. The IDL is used as a specification for methods and data that can be written in and invoked from any language that provides CORBA bindings. Java and C++ are such languages. For a complete description, see the Java Language to OMG IDL Mapping Specification.

Restriction: The generated IDL can be compiled using only an IDL compiler that supports the CORBA 2.3 extensions to IDL.

-always
Forces regeneration even when existing stubs, ties, or IDL are newer than the input class. Valid only when **-iiop** or **-idl** options are present.

-noValueMethods

Ensures that methods and initializers are not included in valuetypes emitted during IDL Generation. These are optional for valuetypes and are otherwise omitted.

Only valid when used with **-idl** option.

-idlModule *<fromJavaPackage[.class]> <toIDLModule>*

Specifies IDLEntity package mapping. For example: `-idlModule foo.bar my::real::idlmod`.

Only valid when used with **-idl** option.

-idlFile *<fromJavaPackage[.class]> <toIDLModule>*

Specifies IDLEntity file mapping. For example: `-idlFile test.pkg.X TEST16.idl`.

Only valid when used with **-idl** option.

More Information

For more detailed information on the rmic compiler, refer to the RMIC tool page (Solaris Version/Windows version).

The idlj compiler

Reference information on the idlj compiler.

Purpose

The idlj compiler generates Java bindings from an IDL file. This compiler supports the CORBA Objects By Value feature, which is required for interoperation with RMI-IIOP. It is written in Java, and so can run on any platform.

More Information

For usage information on the idlj compiler, refer to the IDL-to-Java Compiler User's Guide.

Making RMI programs use IIOP

A general guide to converting an RMI application to use RMI-IIOP.

To use these instructions, your application must already use RMI.

1. If you are using the RMI registry for naming services, you must switch to CosNaming:
   a. In both your client and server code, create an InitialContext for JNDI. For a Java application use the following code:

      ```
      import javax.naming.*;
      ...
      Context ic = new InitialContext();
      ```

      For an applet, use this alternative code:

      ```
      import java.util.*;
      import javax.naming.*;
      ...
      Hashtable env = new Hashtable();
      env.put("java.naming.applet", this);
      Context ic = new InitialContext(env);
      ```

   b. Modify all uses of RMI registry lookup(), bind(), and rebind() to use JNDI lookup(), bind(), and rebind() instead. Instead of:

      ```
      import java.rmi.*;
      ...
      Naming.rebind("MyObject", myObj);
      ```

      use:

      ```
      import javax.naming.*;
      ...
      ic.rebind("MyObject", myObj);
      ```

2. If you are not using the RMI registry for naming services, you must have some other way of bootstrapping your initial remote object reference. For example, your server code might be using Java serialization to write an RMI object reference to an ObjectOutputStream and passing this to your client code for deserializing into an RMI stub. When doing this in RMI-IIOP, you must also ensure that object references are connected to an ORB before serialization and after deserialization.
   a. On the server side, use the PortableRemoteObject.toStub() call to obtain a stub, then use writeObject() to serialize this stub to an ObjectOutputStream. If necessary, use Stub.connect() to connect the stub to an ORB before serializing it. For example:

      ```
      org.omg.CORBA.ORB myORB = org.omg.CORBA.ORB.init(new String[0], null);
      Wombat myWombat = new WombatImpl();
      javax.rmi.CORBA.Stub myStub = (javax.rmi.CORBA.Stub)PortableRemoteObject.toStub(myWombat);
      myStub.connect(myORB);
      // myWombat is now connected to myORB.  To connect other objects to the
      // same ORB, use PortableRemoteObject.connect(nextWombat, myWombat);
      FileOutputStream myFile = new FileOutputStream("t.tmp");
      ObjectOutputStream myStream = new ObjectOutputStream(myFile);
      myStream.writeObject(myStub);
      ```

   b. On the client side, use readObject() to deserialize a remote reference to the object from an ObjectInputStream. Before using the deserialized stub to call remote methods, it must be connected to an ORB. For example:

      ```
      FileInputStream myFile = new FileInputStream("t.tmp");
      ObjectInputStream myStream = new ObjectInputStream(myFile);
      Wombat myWombat = (Wombat)myStream.readObject();
      org.omg.CORBA.ORB myORB = org.omg.CORBA.ORB.init(new String[0], null);
      ((javax.rmi.CORBA.Stub)myWombat).connect(myORB);
      // myWombat is now connected to myORB.  To connect other objects to the
      // same ORB, use PortableRemoteObject.connect(nextWombat, myWombat);
      ```

The JNDI approach is much simpler, so it is preferable to use it whenever possible.

3. Either change your remote implementation classes to inherit from javax.rmi.PortableRemoteObject, or explicitly to export implementation objects after creation by calling PortableRemoteObject.exportObject(). For more discussion on this topic, read "Connecting IIOP stubs to the ORB."
4. Change all the places in your code where there is a Java cast of a remote interface to use javax.rmi.PortableRemoteObject.narrow().
5. Do not depend on distributed garbage collection (DGC) or use any of the RMI DGC facilities. Use PortableRemoteObject.unexportObject() to make the ORB release its references to an exported object that is no longer in use.
6. Regenerate the RMI stubs and ties using the rmic command with the **-iiop** option. This will produce stub and tie files with the following names:

   _<*implementationName*>_Tie.class
   _<*interfaceName*>_Stub.class
7. Before starting the server, start the CosNaming server (in its own process) using the `tnameserv` command The CosNaming server uses the default port number of 2809. If you want to use a different port number, use the **-ORBInitialPort** parameter.
8. When starting client and server applications, you must specify some system properties. When running an application, you can specify properties on the command line:

   ```
   java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
        -Djava.naming.provider.url=iiop://<hostname>:2809
        <appl_class>
   ```

   When running an applet, you can specify properties in the applet tag:

   ```
   java.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
   java.naming.provider.url=iiop://<hostname>:2809
   ```

   The examples uses the default name service port number of 2809. If you specify a different port in the previous step, you need to use the same port number in the provider URL here. The <*hostname*> in the provider URL is the host name that was used to start the CosNaming server in the previous step.
9. If the client is an applet, you must specify some properties in the applet tag. For example:

   ```
   java.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
   java.naming.provider.url=iiop://<hostname>:2809
   ```

   This example uses the default name service port number of 2809. If you specify a different port in the previous step, you need to use the same port number in the provider URL here. The <*hostname*> in the provider URL is the host name that was used to start the CosNaming server in the previous step.

Your application can now communicate with CORBA objects using RMI-IIOP.

Connecting IIOP stubs to the ORB

When your application uses IIOP stubs, as opposed to JRMP stubs, you must properly connect the IIOP stubs with the ORB before invoking operations on the IIOP stubs (this is not necessary with JRMP stubs). This section discusses the extra 'connect' step required for the IIOP stub case.

The PortableRemoteObject.exportObject() call only creates a tie object and caches it for future usage. The created tie does not have a delegate or an ORB associated. This is known as explicit invocation.

The PortableRemoteObject.exportObject() happens automatically when the servant instance is created. The servant instance is created when a PortableRemoteObject constructor is called as a base class. This is known as implicit invocation.

Later, when the application calls PortableRemoteObject.toStub(), the ORB creates the corresponding Stub object and associates it with the cached Tie object. But because the Tie is not connected and does not have a delegate, the newly created stub also does not have a delegate or ORB.

The delegate is set for the stub only when the application calls Stub.connect(orb). Thus, any operations on the stub made before the ORB connection is made will fail.

The Java Language to OMG IDL Mapping Specification says this about the Stub.connect() method:

> "The connect method makes the stub ready for remote communication using the specified ORB object orb. Connection normally happens implicitly when the stub is received or sent as an argument on a remote method call, but it is sometimes useful to do this by making an explicit call (e.g., following deserialization). If the stub is already connected to orb (has a delegate set for orb), then connect takes no action. If the stub is connected to some other ORB, then a RemoteException is thrown. Otherwise, a delegate is created for this stub and the ORB object orb."

For servants that are not POA-activated, Stub.connect(orb) is necessary as a required setup.

Restrictions when running RMI programs over IIOP

A list of limitations when running RMI programs over IIOP.

To make existing RMI programs run over IIOP, observe the following restrictions.

- Make sure all constant definitions in remote interfaces are of primitive types or String and evaluated at compile time.
- Do not use Java names that conflict with IDL mangled names generated by the Java-to-IDL mapping rules. See section 28.3.2 of the Java Language to OMG IDL Mapping Specification for more information.
- Do not inherit the same method name into a remote interface more than once from different base remote interfaces.
- Be careful when using names that are identical other than their case. The use of a type name and a variable of that type with a name that differs from the type name in case only is supported. Most other combinations of names that are identical other than their case are not supported.
- Do not depend on runtime sharing of object references to be preserved exactly when transmitting object references across IIOP. Runtime sharing of other objects is preserved correctly.
- Do not use the following features of RMI, which do not work in RMI-IIOP:
  - RMISocketFactory
  - UnicastRemoteObject

– Unreferenced
– The Distributed Garbage Collector (DGC) interfaces

Other things you should know

Servers need to be thread safe

Because remote method invocations on the same remote object might execute concurrently, a remote object implementation must be thread-safe.

Interoperating with other ORBs

RMI-IIOP should interoperate with other ORBs that support the CORBA 2.3 specification. It will not interoperate with older ORBs, because older ORBs cannot handle the IIOP encodings for Objects By Value. This support is needed to send RMI value classes (including strings) over IIOP.

Note: Although ORBs written in different languages should be able to interoperate, the Java ORB has not been fully tested with other vendors' ORBs.

When do I use UnicastRemoteObject vs PortableRemoteObject?

Use UnicastRemoteObject as the superclass for the object implementation in RMI programming. Use PortableRemoteObject in RMI-IIOP programming. If PortableRemoteObject is used, you can switch the transport protocol to either JRMP or IIOP during runtime.

Known problems

- JNDI 1.1 does not support *java.naming.factory.initial*=com.sun.jndi.cosnaming.CNCtxFactory as an Applet parameter. Instead, it must be explicitly passed as a property to the InitialContext constructor. This capability is supported in JNDI 1.2.
- When running the Naming Service on Unix based platforms, you must use a port number greater than 1024. The default port is 2809, so should cause no problems.

Appendix C. Security Guide

An overview of the security features in the IBM SDK for Java.

Preface

The security components described in this User Guide are shipped with the SDK and are not extensions. They provide a wide range of security services through standard Java(TM) APIs (except iKeyman). The security components contain the IBM(R) implementation of various security algorithms and mechanisms. IBM does not provide support for any of the IBM Java security components when used with a non-IBM JVM or with non-IBM security providers when used with the IBM JVM.

The IBM SDK also provides a FIPS 140-2 certified cryptographic module, IBMJCEFIPS, implemented as a JCE provider. Applications can comply with the FIPS 140-2 requirements by using the IBMJCEFIPS module.

The CertPath component provides PKIX-compliant certification path building and validation.

The JGSS component provides a generic API that can be plugged in by different security mechanisms. IBM JGSS uses Kerberos V5 as the default mechanism for authentication and secure communication.

The JAAS component provides a means for principal-based authentication and authorization.

The JCE framework has two providers: IBMJCE is the pre-registered default provider; IBMJCEFIPS is optional.

JSSE is the Java implementation of the SSL and TLS protocols. The JSSE pre-registered default provider is IBMJSSE2.

IBM Java Simple Authentication and Security Layer, or SASL, is an Internet standard (RFC 2222) that specifies a protocol for authentication and optional establishment of a security layer between client and server applications.

The Java security configuration file does not refer to the Sun provider. The IBM JCE provider has replaced the Sun provider. The JCE supplies all the signature handling message digest algorithms that were previously supplied by the Sun provider. It also supplies the IBM secure random number generator, IBMSecureRandom, which is a real Random Number Generator. SHA1PRNG is a Pseudo Random Number Generator and is supplied for code compatibility. SHA1PRNG is not guaranteed to produce the same output as the SUN SHA1PRNG.

In the IBM SDK v1.4.1, the following options were added to the **java.security.debug** property to help you debug Java Cryptography Architecture (JCA)-related problems:

provider
: Displays each provider request and load, provider add and provider remove. It also displays the related exception when a provider load fails.

algorithm
: Displays each algorithm request, which provider has supplied the algorithm and the implementing class name.

:stack You can append this option to either of **algorithm** - or **provider**. When you request an algorithm, a stack trace is displayed. Use this stack trace to

determine the code that has requested the algorithm. This option also prints the stack trace for exceptions that are swallowed or converted.

:thread
Adds the thread id to all debug message lines. You can use this option together with all the other debug options.

An example of a valid option string is ″provider, algorithm:stack″.

In this guide, you will see a 'What's new' section for each component. This information is provided to help you with migration.

General information about IBM security providers

Overview of the security providers tested with the IBM SDK.

The IBM SDK v5.0 has been tested with the following default security providers:

- security.provider.1=com.ibm.jsse2.IBMJSSEProvider2
- security.provider.2=com.ibm.crypto.provider.IBMJCE
- security.provider.3=com.ibm.security.jgss.IBMJGSSProvider
- security.provider.4=com.ibm.security.cert.IBMCertPath
- security.provider.5=com.ibm.security.sasl.IBMSASL

You can add other IBM security providers either statically or from within your Java application's code. To add a new provider statically, edit a java security properties file (for example, java.security). To add a new provider from your application's code, use the methods of the java.security.Security class (for example, `java.security.Security.addProvider()`).

You can also add this IBM security provider, com.ibm.crypto.fips.provider.IBMJCEFIPS.

Note that code written for the IBMJSSE Provider might not compile or execute in exactly the same way for IBMJSSE2. For details, see "IBMJSSE2 Provider" on page 267.

iKeyman tool

Overview of the iKeyman tool.

The iKeyman utility is a tool for managing your digital certificates. With iKeyman, you can:

- Create a new key database or a test digital certificate
- Add CA roots to your database
- Copy certificates from one database to another
- Request and receive a digital certificate from a CA
- Set default keys, and change passwords

What's new?

History of changes to the iKeyman tool.

There are no changes for v5.0 over v1.4.2.

There are no changes in v1.4.2 over v1.4.1.

The following change was added in v1.4.1:

- An iKeyman wrapper that invokes the correct tool class was added.

Documentation

Available documentation for the iKeyman tool.

The *iKeyman User Guide* is on the developerWorks Web site, at http://www.ibm.com/developerworks/java/jdk/security/index.html.

Java Authentication and Authorization Service (JAAS) V2.0

The Sun Microsystems Java 2 platform provides a means to enforce access controls based on *where* code came from and *who signed* it. These access controls are needed because of the distributed nature of the Java platform where, for example, a remote applet can be downloaded over a public network and then run locally.

However, before SDK v1.4.0, the Java 2 platform did not provide a way to enforce similar access controls based on *who runs* the code. To provide this type of access control, the Java 2 security architecture requires the following:

- Additional support for authentication (determining who is actually running the code)
- Extensions to the existing authorization components to enforce new access controls based on who was authenticated

The Java Authentication and Authorization Service (JAAS) framework provides these enhancements.

For a general overview of JAAS, see the Sun Web site: http://java.sun.com/products/jaas.

Differences between IBM and Sun versions of JAAS

The IBM version of JAAS differs from the Sun version in the following ways:

- The com.sun.* packages are reimplemented by IBM and renamed com.ibm.* packages.

What's new?

There are no changes to JAAS in v5.0.

There are no changes in v1.4.2 over v1.4.1.

There were no changes in v1.4.1 over v1.4.0. However, the following changes were added in v1.4.0.

What has been added to the IBM 32-bit SDK for Linux on Intel architecture

???????? Audrey - why is this IA32 only???

The original release of JAAS for Linux and the Java 2 Platform included the following login module and principal classes:

- com.ibm.security.auth.module.LinuxLoginModule
- com.ibm.security.auth.LinuxPrincipal

- com.ibm.security.auth.LinuxNumericGroupPrincipal
- com.ibm.security.auth.LinuxNumericUserPrincipal

These original platform-dependent principal classes will be replaced by a set of platform-independent principal classes in future releases of JAAS for Linux. To ease migration, this version of JAAS contains the original set as well as the new set of principal classes. Additional principal classes have been included to facilitate the writing of new login modules.

You are encouraged to use the new set of principals when developing applications that use JAAS. Previously developed applications will be compatible with this version as well as future versions of JAAS released for the SDK v1.4.0.

If migrating applications to the new set of principals is desired, then most changes encountered will be in JAAS policy and configuration files rather than in the applications. Refer to the following table for guidance.

Table 26. New class names

Original class	**Replaced by**
LinuxPrincipal	UsernamePrincipal
LinuxNumericGroupPrincipal	GroupIDPrincipal PrimaryGroupIDPrincipal
LinuxNumericUserPrincipal	UserIDPrincipal
n/a	DomainPrincipal
n/a	DomainIDPrincipal
n/a	ServerPrincipal
n/a	WkstationPrincipal
LinuxLoginModule	LinuxLoginModule2000

Principal classes are found in the com.ibm.security.auth package while the login module is found in the com.ibm.security.auth.module package. Check the JAAS API documentation (javadoc) for more information on the new principal classes.

For example, this JAAS policy grant block:

```
grant Principal com.ibm.security.auth.LinuxPrincipal "bob",
      Principal com.ibm.security.auth.LinuxNumericUserPrincipal
           "727",
      Principal com.ibm.security.auth.LinuxNumericGroupPrincipal
           "12" {
   permission java.util.PropertyPermission "java.home", "read";
};
```

would be replaced by:

```
grant Principal com.ibm.security.auth.UsernamePrincipal "bob",
      Principal com.ibm.security.auth.UserIDPrincipal "727",
      Principal com.ibm.security.auth.GroupIDPrincipal "12" {
   permission java.util.PropertyPermission "java.home", "read";
};
```

Documentation

For detailed information, including API documentation and samples, see the developerWorks Web site at http://www.ibm.com/developerworks/java/jdk/security/index.html.

Java Certification Path (CertPath)

The Java Certification Path API provides interfaces and abstract classes for creating, building, and validating certification paths (also known as ″certificate chains″).

Differences between IBM and Sun versions of CertPath

The IBM CertPath classes differ from the Sun version in the following ways:

- The IBM CertPath provider is in the package com.ibm.security.cert.
- The IBM CertPath provider is called ″IBMCertPath″. Sun does not have a separate provider for CertPath; CertPath is already supported by the ″SUN″ provider.
- To enable CRL Distribution Points extension checking, use the system property **com.ibm.security.enableCRLDP**. The system property used by the Sun version is **com.sun.security.enableCRLDP**.
- When checking the certificate's CRL Distribution Points extension, Sun's version retrieves the CRL only if the CRL location is specified as an HTTP URL value inside the extension. The IBM provider recognizes both HTTP and LDAP URLs.

What's new?

The following changes have been added in v5.0:

- Support of checking a certificate's revocation status based on On-Line Certificate Status Protocol (OCSP) has been added.
- A new constructor and a public API in TrustAnchor class have been added:
  - `public TrustAnchor(X500Principal caPrincipal, PublicKey pubKey, byte[] nameConstraints);`
  - `public final X500Principal getCA();`
- Four new public APIs in X509CertSelector have been added:
  - `public X500Principal getIssuer();`
  - `public void setIssuer(X500Principal issuer);`
  - `public X500Principal getSubject();`
  - `public void setSubject(X500Principal subject);`
- Three new public APIs in X509CRLSelector have been added:
  - `public void setIssuers(Collection issuers);`
  - `public void addIssuer(X500Principal issuer);`
  - `public Collection getIssuers();`
- The PolicyQualifier class has been changed to be non-final and its public APIs have changed to be final.

The following changes were added in v1.4.2:

- The performance of the IBM CertPath provider has been improved.
- Limited support for the CRL Distribution Points extension has been added.
- IBM LDAP CertStore provides caching to cache lookups.

The following changes were added in v1.4.1 SR1:

- The trusted certificate that acts as TrustAnchor can be an X.509 v1 certificate.
- When you specify the certificate's subject or issuer name as a String in X509CertSelector, the search for a matched certificate mechanism checks only the name value and ignores the tag type.

There were no changes in v1.4.1 over v1.4.0.

The following changes were added in v1.4.0:

- Certificates from CertificatePair entry can be retrieved from LDAP type certstore.
- The framework package name was changed from javax.security.cert to java.security.cert. However, the old framework package is still supported.

Documentation

For detailed information, including API documentation and samples, see the developerWorks Web site, at http://www.ibm.com/developerworks/java/jdk/security/index.html.

Java Cryptography Extension (JCE)

The Java Cryptography Extension (JCE) provides a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. The software also supports secure streams and sealed objects. JCE supplements the Java 2 platform, which already includes interfaces and implementations of message digests and digital signatures.

You can obtain unrestricted jurisdiction policy files from http://www.ibm.com/developerworks/java/jdk/security/index.html.

The v1.4.2 unrestricted (and restricted) jurisdiction policy files are suitable for use with v5.0. The v1.4.1 files are not suitable.

Differences between IBM and Sun versions of JCE

The com.sun.* packages are reimplemented by IBM and renamed com.ibm.* packages.

The IBM version of JCE differs from the Sun version in the following ways:

- The com.sun.crypto.* packages are reimplemented by IBM and renamed com.ibm.crypto.* packages.
- The IBM JCE provider replaces the Sun providers sun.security.provider.Sun, com.sun.rsajca.Provider, and com.sun.crypto.provider.SunJCE.
- IBM provides more algorithms than Sun does:
  - **Cipher algorithms**
    - AES
    - Blowfish
    - DES
    - Mars
    - ARCFOUR
    - PBE with MD2 and DES
    - PBE with MD2 and Triple DES
    - PBE with MD2 and RC2
    - PBE with MD5 and DES
    - PBE with MD5 and Triple DES
    - PBE with MD5 and RC2

- PBE with SHA1 and DES
- PBE with SHA1 and TripleDES
- PBE with SHA1 and RC2
- PBE with SHA1 and 40-bit RC2
- PBE with SHA1 and 128-bit RC2
- PBE with SHA1 and 40-bit RC4
- PBE with SHA1 and 128-bit RC4
- PBE with SHA1 and 2-key Triple DES
- PBE with SHA1 and 3-key Triple DES
- RC2
- RC4
- RSA encryption/decryption
- RSA encryption/decryption with OAEP Padding
- Seal
- Triple DES

– **Signature algorithms**
  - SHA1 with RSA, SHA2 with RSA, SHA3 with RSA, SHA5 with RSA, MD5 with RSA, MD2 with RSA signatures
  - SHA1 with DSA signature

– **Message digest algorithms**
  - SHA1
  - SHA2
  - SHA3
  - SHA5
  - MD5
  - MD2

– **Message authentication code (MAC)**
  - Hmac/SHA1
  - Hmac/MD5
  - Hmac/SHA2
  - Hmac/SHA3
  - Hmac/SHA5

– **Key agreement algorithm**
  - DiffieHellman

– **Random number generation algorithms**
  - IBMSecureRandom
  - IBM SHA1PRNG

– **Key Store**
  - JCEKS
  - JKS
  - PKCS12KS

What's new?

The following changes are made in v5.0:

- RSA with OAEP Padding is added

- SHA2withRSA, SHA3withRSA and SHA5withRSA signature algorithms are added
- HmacSHA2, HmacSHA3, HmacSH5 MAC algorithms are added.
- ARCFOUR encryption algorithm is added

The following changes were made in v1.4.2:

- SHA2, SHA3 and SHA5 algorithms were added for hashing
- SHA1PRNG algorithm was added for generating pseudo random numbers

There were no changes in v1.4.1 from v1.4.0.

The following changes were made in v1.4.0:

- AES cipher algorithm has been added.
- Strong cryptography is the default, unlimited cryptography is available.
- Provider authentication of the JCE framework no longer required.
- JCE is now shipped with the Java SDK v1.4 on all platforms.

Documentation

For detailed information, including API documentation and samples, see the developerWorks Web site at http://www.ibm.com/developerworks/java/jdk/security/index.html.

Java Generic Security Service (JGSS)

The Java Generic Security Service (JGSS) API provides secure exchange of messages between communicating applications.

The JGSS is an API framework that has Kerberos V5 as the underlying default security mechanism. The API is a standardized abstract interface under which you can plug different security mechanisms that are based on private-key, public-key, and other security technologies. JGSS shields secure applications from the complexities and peculiarities of the different underlying security mechanisms. JGSS provides identity and message origin authentication, message integrity, and message confidentiality. JGSS also features an optional Java Authentication and Authorization Service (JAAS) Kerberos login interface, and authorization checks. JAAS augments the access control features of Java 2, which is based on CodeSource with access controls based on authenticated principal identities.

Differences between IBM and Sun versions of JGSS

The IBM version of JGSS differs from the Sun version in the following ways:

- The com.sun.* packages are reimplemented by IBM and renamed com.ibm.* packages.
- The format of the parameters passed to the Java tools kinit, ktab, and klist is different from Sun's equivalent tools.

What's new?

The following change is added in v5.0 Service Refresh 1:

AES is now a supported algorithm type

These additional algorthims can be set in the krb5.conf file under [libdefault] as follows:

```
default_tkt_enctypes = aes128-cts-hmac-sha1-96
default_tkt_enctypes = aes256-cts-hmac-sha1-96
default_tgs_enctypes = aes128-cts-hmac-sha1-96
default_tgs_enctypes = aes256-cts-hmac-sha1-96

default_checksum   = hmac-sha1-96-aes128
default_checksum   = hmac-sha1-96-aes256
```

The following changes are added in v5.0:

TCP or UDP Preference Configuration

JSE now supports the use of the **udp_preference_limit** property in the Kerberos configuration file (krb5.ini). When sending a message to the KDC, the JSE Kerberos library will use TCP if the size of the message is above **udp_preference_list**. If the message is smaller than **udp_preference_list**, UDP will be tried up to three times. If the KDC indicates that the request is too big, the JSE Kerberos library will use TCP.

IPv6 support in Kerberos

JSE now supports IPv6 addresses in Kerberos tickets. Before J2SE 5, only IPv4 addresses were supported in tickets.

TGT Renewals

The Java Authentication and Authorization Server (JAAS) Kerberos login module in v5.0, Krb5LoginModule, now supports Ticket Granting Ticket (TGT) renewal. This support allows long-running services to renew their TGTs automatically without user interaction or requiring the services to restart. With this feature, if Krb5LoginModule obtains an expired ticket from the ticket cache, the TGT will be automatically renewed and be added to the Subject of the caller who requested the ticket. If the ticket cannot be renewed for any reason, Krb5LoginModule will use its configured callback handler to retrieve a username and password to acquire a new TGT.

To use this feature, configure Krb5LoginModule to use the ticket cache and set the newly introduced **renewTGT** option to true. Here is an example of a JAAS login configuration file that requests TGT renewal:

```
server {
  com.ibm.security.auth.module.Krb5LoginModule required
        principal=principal@your_realm
  useDefaultCcache=TRUE
   renewTGT=true;
};
```

Note that if **renewTGT** is set to true, **useDefaultCcache** must also be set to true; otherwise, it results in a configuration error.

The following changes were added in v1.4.2:

Configurable Kerberos Settings

You can provide the name and realm settings for the Kerberos Key Distribution Center (KDC) either from the Kerberos configuration file or by using the system properties files java.security.krb5.kdc and java.security.krb5.realm. You can also specify the boolean option **refreshKrb5Config** in the entry for Krb5LoginModule in the JAAS configuration file. If you set this option to `true`, the configuration values will be refreshed before the login method of the Krb5LoginModule is called.

Support for Slave Kerberos Key Distribution Center

Kerberos uses slave KDCs so that, if the master KDC is unavailable, the

slave KDCs will respond to your requests. In previous releases, Kerberos tried the master KDC only and would give up if there was no response within the default KDC timeout.

Support TCP for Kerberos Key Distribution Center Transport
Kerberos uses UDP transport for ticket requests. In cases where Kerberos tickets exceed the UDP packet size limit, Kerberos supports automatic fallback to TCP. If a Kerberos ticket request using UDP fails and the KDC returns the error code `KRB_ERR_RESPONSE_TOO_BIG`, TCP becomes the transport protocol.

Kerberos Service Ticket in the Subject's Private Credentials
The Kerberos service ticket is stored in the Subject's private credentials. This gives you access to the service ticket so that you can use it outside the JGSS (for example, in native applications or for proprietary uses). In addition, you can reuse the service ticket if the application tries to establish a security context to the same service again. The service ticket should be valid for it to be reusable.

The following change was added in v1.4.1:

- Wrappers have been added for the klist, kinit, and ktab Java tools. These wrappers invoke the relevant tool classes so that you do not have to remember the full package name.

Documentation

For detailed information about JGSS, including API documentation and samples, see the developerWorks Web site, at http://www.ibm.com/developerworks/java/jdk/security/index.html.

IBMJSSE2 Provider

The Java Secure Socket Extension (JSSE) is a Java package that enables secure internet communications. It implements a Java version of SSL (Secure Sockets Layer) and TLS (Transport Layer Security) protocols and includes functions for data encryption, server authentication, message integrity, and optional client authentication.

By abstracting the complex underlying security algorithms and "handshaking" mechanisms, JSSE minimizes the risk of creating subtle but dangerous security vulnerabilities. Also, it simplifies application development by serving as a building block that you can integrate directly into your applications. Using JSSE, you can provide for the secure passage of data between a client and a server running any application protocol (such as HTTP, Telnet, NNTP, and FTP) over TCP/IP.

In v5.0, the IBMJSSE2 Provider, which was introduced in the v1.4.2 JVM, has replaced the IBMJSSE Provider. Although they are nearly equivalent, there are differences between the two providers. See the next section for details.

Differences between the IBMJSSE Provider and the IBMJSSE2 Provider

The now-discontinued IBMJSSE Provider and the IBMJSSE2 Provider differ in the following ways:

- The IBMJSSE2 Provider is called com.ibm.jsse2.IBMJSSEProvider2.
- The HTTPS protocol handler for the IBMJSSE2 Provider is called com.ibm.net.ssl.www2.protocol.Handler. The

com.ibm.net.ssl.internal.www.protocol.Handler and the com.ibm.net.ssl.www.protocol.Handler protocol handlers have been removed.

- The IBMJSSE2 Provider does not support the com.ibm.net.ssl framework. Use the javax.net.ssl framework instead.
- The IBMJSSE2 Provider does not support the SSL version 2 protocol. However, the server side of a JSSE2 connection does accept the SSLv2Hello protocol.
- The AES_256 ciphers require the installation of the JCE Unlimited Strength Jurisdiction Policy. The old IBMJSSE Provider did not use JCE for its cryptographic support and therefore did not require these files.
- The IBMJSSE2 Provider requires a JCE Provider for its cryptography.
- The IBMJSSE2 Provider does not build the server's private key certificate chain from the trusted keystore. The trusted certificates must be added to the server's private key to complete the chain. This is an incompatible change.
- The IBMJSSE2 Provider considers a certificate trusted if you have the private key.
- The HTTPS protocol handler for the IBMJSSE2 Provider performs hostname verification and rejects requests where the host to connect to and the server name from the certificate do not match. A HostnameVerification implementation called com.ibm.jsse2.HostnameVerifierIgnore is provided. com.ibm.jsse2.HostnameVerifierIgnore always accepts the connection even when a mismatch occurs.
- Tracing no longer requires a separate debug jar.
- The class com.ibm.jsse.SSLContext which in IBMJSSE is used to access secure tokens has been removed. Use the hardware crypto support in IBMJSSE2 instead. See the documentation on the developerWorks Web site http://www.ibm.com/developerworks/java/jdk/security/index.html for details.
- The IBMJSSEFIPS Provider has been removed. JSSE FIPS support is supported within the IBMJSSE2 Provider and no separate jar is required. See the documentation on the developerWorks Web site http://www.ibm.com/developerworks/java/jdk/security/index.html for instructions how to set up JSSE to run in FIPS mode.

Differences between the IBMJSSE2 Provider and Sun's version of JSSE

The IBMJSSE2 Provider differs from the Sun JSSE in the following ways:

- The IBM JSSE Provider is called com.ibm.jsse2.IBMJSSEProvider2.
- The IBM KeyManagerFactory is called IbmX509.
- The IBM TrustManagerFactory is called IbmX509 or IbmPKIX.
- The IBM HTTPS protocol handler is called com.ibm.net.ssl.www2.protocol.Handler.
- IBMJSSE2 does not support the com.sun.net.ssl framework; use the javax.net.ssl framework instead.
- You can use PKIK revocation checking by setting the system property *com.ibm.jsse2.checkRevocation* to "true".
- The IBM implementation supports the following protocols for the engine class SSLContext, for the api setEnabledProtocols in the SSLSocket, and for SSLServerSocket classes:
  - SSL
  - SSLv3
  - TLS

- TLSv1
- SSL_TLS

The IBM implementation *does not* support the ″SSLv2Hello″ protocol. The IBM implementation supports the SSL v2 protocol. You can use the IBM SSLContext `getInstance()` factory method to control which protocols are enabled for an SSL connection. Using SSLContext's `getInstance()` or the `setEnabledProtocols()` methods provides the same result. With Sun's JSSE, the protocol is controlled through `setEnabledProtocols()`.

- IBM and Sun support different cipher suites.
- The IBM JSSE TrustManager does not allow anonymous ciphers. To handshake with an anonymous cipher, a custom TrustManager that allows anonymous ciphers must be provided.
- When a null KeyManager is passed to SSLContext, the IBM JSSE KeyManagerFactory implemention will check system properties, then jssecacerts, if it exists, and finally uses the cacerts file to find the key material. Sun's JSSE creates an empty KeyManager.
- The IBM JSSE X509TrustManager and X509KeyManager throws an exception if the TrustStore or KeyStore specified by the system properties does not exist, if the password is incorrect, or if the keystore type is inappropriate for the actual keystore. Sun's X509TrustManager creates a default TrustManager or KeyManager with an empty keystore.
- The IBM JSSE implementation verifies the entire server or client certificate chain, including trusted certificates. For example, if a trusted certificate has expired, the handshake fails, even though the expired certificate is trusted. Sun's JSSE verifies the certificate chain up to the trusted certificate. Verification stops when it reaches a trusted certificate and the trusted certificate and beyond are not verified.
- The IBM JSSE implementation returns the same set of supported ciphers for the methods `getDefaultCiphersSuites()` and `getSupportedCipherSuites()`. Sun's JSSE `getDefaultCipherSuites()` returns the list of cipher suites that provide confidentiality protection and server authentication (that is, no anonymous cipher suites). Sun's `getEnabledCipherSuites()` returns the entire list of cipher suites that Sun supports.
- For Sun's implementation, DSA server certificates can use only *_DH*_* cipher suites. For the IBM implementation, if the server has a DSA certificate only and only RSA* ciphers are enabled, the connection succeeds with an RSA cipher. DSA will be used for authentication and ephemeral RSA will be used for the key exchange.
- To use a hardware keystore or truststore with IBM's hardware crypto provider by means of system properties, set the **javax.net.ssl.keyStoreType** and **javax.net.ssl.trustStoreType** system properties, respectively, to ″PKCS11IMPLKS″. For the Sun implementation, the system property is set to ″PKCS11″.
- The IBM JSSE Provider can be enabled to run in FIPS mode. The Sun JSSE cannot.

What's new?

The following changes have been added in v5.0

- The IBMJSSE Provider is removed for v5.0. Use the IBMJSSE2 Provider instead.
- The IBMJSSE2 implementation now uses the Java Cryptography Extension (JCE) for all its cryptographic algorithms. In v1.4.2, only the IBMJCE providers were supported.

- The IBMJSSE2 implementation now supports any hardware crypto provider. This support will require applications that ran in v1.4.2 using the IBMPKCS11Impl provider to use a configuration file in order to run successfully.
- SSLEngine (non-blocking I/O) allows SSL/TLS applications to choose their own I/O and compute models.
- Enhanced TrustManager support HTTP/HTTPS enhancements.
- New and updated Methods and Classes.
- Kerberos cipher suites are available, if supported by the operating system.

The IBMJSSE2 Provider was new for v1.4.2.

Documentation

For detailed information, including API documentation and samples, see http://www.ibm.com/developerworks/java/jdk/security/index.html.

IBMPKCS11Impl Provider

The IBMPKCS11Impl Provider is not supported on the HP-UX 11i.

The IBMPKCS11Impl Provider uses the Java Cryptography Extension (JCE) and Java Cryptography Architecture (JCA) frameworks to add the ability to use hardware cryptography through the Public Key Cryptogaphic Standards #11 (PKCS #11) standard. This provider takes advantage of hardware cryptography within the existing JCE architecture and gives Java programmers the significant security and performance advantages of hardware cryptography with minimal changes to existing Java applications. Because the complexities of hardware cryptography are handled within the normal JCE, advanced security and performance using hardware cryptographic devices is available readily.

PKCS#11 is a standard that provides a common application interface to cryptographic services on various platforms through several hardware cryptographic devices. See the IBMPKCS11Impl provider user guide for a list of supported devices.

Differences between IBM and Sun versions of IBMPKCS11Impl

The most significant difference between the Sun PKCS11 provider and the IBM PKCS11Impl provider is in the area of keystore. Sun has a keystore named PKCS11 and IBM has one called IBMPKCS11KS. Sun requires that all trusted certificates have the attribute **CKA_TRUSTED** set to true. The IBM keystore assumes that any certificates on the device are trusted. So, this assumption should allow IBM's keystore to work with data that was saved using the Sun PKCS11 provider keystore, but not the other way around.

What's new?

The following changes were made in v5.0.

IBMPKCS11Impl has been updated to allow more algorithms and to allow the Sun 5.0 methods of initialization of the provider. The new algorithms are:

- AES
- Diffie-Hellman
- RC4, also known as ArcFour
- Blowfish

- SHA-256
- SHA-384
- SHA-512
- SHA256withRSA
- SHA384withRSA
- SHA512withRSA
- HmacMD5
- HmacSHA1
- HmacSHA256
- HmacSHA384
- HmacSHA512

In v5.0, the ability to pass in a configuration file to the provider is added. This configuration file can contain a significant amount of information about the device; for example, what it should or should not do. After the provider is created, the application can log in to the card in different ways. Some devices allow you to perform some cryptographic functions without logging into the device. The v1.4.2 ways to initialize the device still work. However, you can no longer have more than one of these providers at a time. Instead, with this release, you can initialize more than one IBMPKCS11Impl provider using the 5.0 configuration file and login methods.

The classes DESPKCS11KeyParameterSpec and DESedePKCS11KeyParameterSpec have been deprecated. Use the GeneralPKCS11KeyParameterSpec class for all symmetric key types (for instance, DES, DESede, AES, RC4, Blowfish).

The IBMPKCS11Impl Provider was new for v1.4.2.

Documentation

For detailed information, including API documentation, see the developerWorks Web site at http://www.ibm.com/developerworks/java/jdk/security/index.html.

IBMJCEFIPS Provider

The IBM Java JCE (Java Cryptographic Extension) FIPS Provider (IBMJCEFIPS) for multi-platforms is a scalable, multi-purpose cryptographic module that supports FIPS-approved cryptographic operations through Java APIs.

The IBMJCEFIPS includes the following Federal Information Processing Standards (FIPS) 140-2 [Level 1] compliant components:

- IBMJCEFIPS for Solaris,
- IBMJCEFIPS for HP
- IBMJCEFIPS for Windows
- IBMJCEFIPS for z/OS
- IBMJCEFIPS for AS/400
- IBMJCEFIPS for Linux (Red Hat and SUSE)

To meet the requirements specified in the FIPS publication 140-2, the encryption algorithms used by the IBMJCEFIPS Provider are isolated into the IBMJCEFIPS Provider cryptographic module, which you can access using the product code from the Java JCE framework APIs. Because the IBMJCEFIPS Provider uses the cryptographic module in an approved manner, the product complies with the FIPS

140-2 requirements.

Type	Algorithm	Specification
Symmetric Cipher	AES (ECB, CBC, OFB, CFB and PCBC)	FIPS 197
Symmetric Cipher	Triple DES (ECB, CBC, OFB, CFB and PCBC)	FIPS 46-3
Message Digest	SHA1 SHA-256 SHA-384 SHA-512 HMAC-SHA1	FIPS 180-2 FIPS 198a
Random Number Generator	FIPS 186-2 Appendix 3.1	FIPS 186-2
Digital Signature	DSA (512 - 1024)	FIPS 186-2
Digital Signature	RSA (512 – 2048)	FIPS 186-2

In addition, the IBMJCEFIPS supports the following unapproved algorithms:

Type	Algorithm	Specification
Asymmetric Cipher	RSA	PKCS#1
Key Agreement	Diffie-Hellman	PKCS #3 (Allowed in Approved mode)
Digital Signature	DSAforSSL	Allowed for use within the TLS protocol
Digital Signature	RSAforSSL	Allowed for use within the TLS protocol
Message Digest	MD5	FIPS 180-2
Random Number Generation	Universal Software Based Random Number Generator	Available upon request from IBM. Patented by IBM, EC Pat. No. EP1081591A2, U.S. pat. Pend.

Important: The com.ibm.crypto.fips.provider.IBMJCEFIPS class does not include a keystore (such as JKS or JCEKS) because of FIPS requirements and algorithms. Therefore, if you are using com.ibm.crypto.fips.provider.IBMJCEFIPS and require JKS, you must specify the com.ibm.crypto.provider.IBMJCE in the provider list.

For more detailed information on the FIPS certified provider IBMJCEFIPS, see the *IBM Java JCE FIPS 140-2 Cryptographic Module Security Policy*. For usage information and details of the API, see the *IBM Java JCE FIPS (IBMJCEFIPS) Cryptographic Module API* document. These documents are available at http://www.ibm.com/developerworks/java/jdk/security/index.html.

Differences between IBM and Sun versions of IBMJCEFIPS

Sun does not provide IBMJCEFIPS.

Documentation

For detailed information, including API documentation and Security Policy, see the developerWorks Web site, at http://www.ibm.com/developerworks/java/jdk/security/index.html.

IBM SASL Provider

Simple Authentication and Security Layer, or SASL, is an Internet standard (RFC 2222) that specifies a protocol for authentication and optional establishment of a security layer between client and server applications. SASL defines how authentication data is to be exchanged but does not itself specify the contents of that data. It is a framework into which specific authentication mechanisms that specify the contents and semantics of the authentication data can fit.

The Java SASL API defines classes and interfaces for applications that use SASL mechanisms. It is defined to be mechanism-neutral: the application that uses the API need not be hardwired into using any particular SASL mechanism. The API supports both client and server applications. It allows applications to select the mechanism to use based on desired security features, such as whether they are susceptible to passive dictionary attacks or whether they accept anonymous authentication. The Java SASL API also allows developers to use their own, custom SASL mechanisms. SASL mechanisms are installed by using the Java Cryptography Architecture (JCA).

The IBMSASL provider supports the following client and server mechanisms.

Client mechanisms

- PLAIN (RFC 2595). This mechanism supports cleartext username/password authentication.
- CRAM-MD5 (RFC 2195). This mechanism supports a hashed username/password authentication scheme.
- DIGEST-MD5 (RFC 2831). This mechanism defines how HTTP Digest Authentication can be used as a SASL mechanism.
- GSSAPI (RFC 2222). This mechanism uses the GSSAPI for obtaining authentication information. It supports Kerberos v5 authentication.
- EXTERNAL (RFC 2222). This mechanism obtains authentication information from an external channel (such as TLS or IPsec).

Server mechanisms

- CRAM-MD5
- DIGEST-MD5
- GSSAPI (Kerberos v5)

Differences between Sun and IBM SASL Provider

Only the package names, for example com.ibm.security.sasl, and the provider name are different from the Sun Implementation: com.ibm.security.sasl.IBMSASL.

What's new

The IBM SASL Provider is new for v5.0

Documentation

Detailed information, including API documentation and samples, is on the developerWorks Web site, at http://www.ibm.com/developerworks/java/jdk/security/index.html.

Key Certificate Management utilities

Uses of the Key Certificate Management utilities.

The Key Certificate Management utilities make up a set of packages used to:

- Access keys and certificates stored in any format
- Extract information from a KeyStore, given a Subject Key Identifier (SKI) and a set of certificate generation APIs, to create a self-signed certificate
- Generate a CertificateRequest
- Obtain a certificate signed by a CA

The Key Certificate Management utilities can:

- Generate a CertificateRequest, and submit the request to a CA using the Java PKI to sign a certificate and then receive the signed certificate
- Generate a PKCS10 request
- Generate a Self-Signed Certificate
- Revoke a signed certificate from a CA using the Java PKI
- Import certificates from the input stream to the KeyStore or export certificates from the KeyStore to the output stream
- Copy a keystore from one keystore format to another keystore format.
- Extract information from a KeyStore given a Subject Key Identifier

The Subject Key Identifier is specified in RFC 3820, Section 4.2.1.2, http://www.faqs.org/rfcs/rfc3820.html.

What's new

The Key Certificate Management utilities are new for Version 5.0, Service Refresh 1.

Documentation

The *Key Certificate Management How-to Guide* and Javadoc are on the developerWorks Web site, at http://www.ibm.com/developerworks/java/jdk/security/index.html.

Index

Special characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

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 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 10504-1758 U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

- IBM World Trade Asia Corporation Licensing
  2-31 Roppongi 3-chome, Minato-ku
  Tokyo 106-0032, Japan

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 information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information 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.

Licensees of this program who wish to have information about it for the purpose of enabling (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

- JIMMAIL@uk.ibm.com
  [Hursley Java Technology Center (JTC) contact]

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

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

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

Trademarks

IBM, iSeries, pSeries, and zSeries are trademarks or registered trademarks of International Business Machines Corporation in the United States, or other countries, or both.

Intel is a trademark of Intel Corporation in the United States, other countries, or both.

IBM is a trademark of International Business Machines Corporation in the United States, or other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

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

Intel and Itanium are trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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