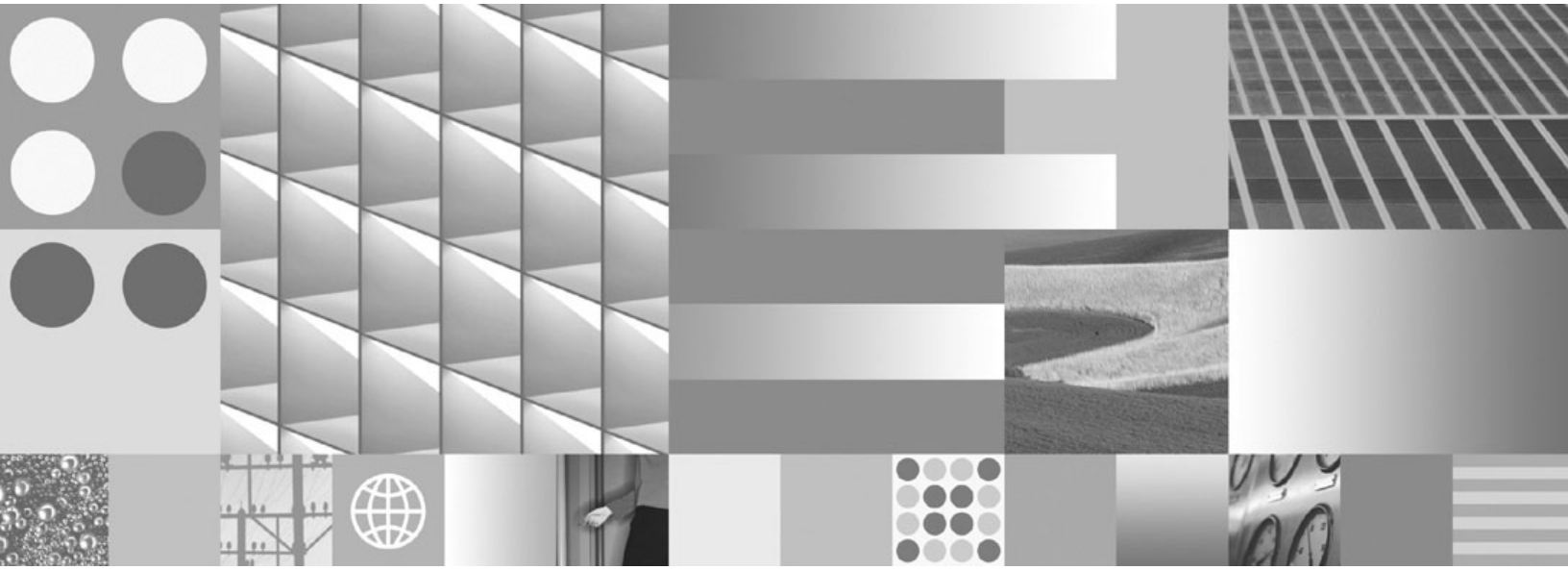
IBM SDK for z/OS platforms, Java Technology Edition

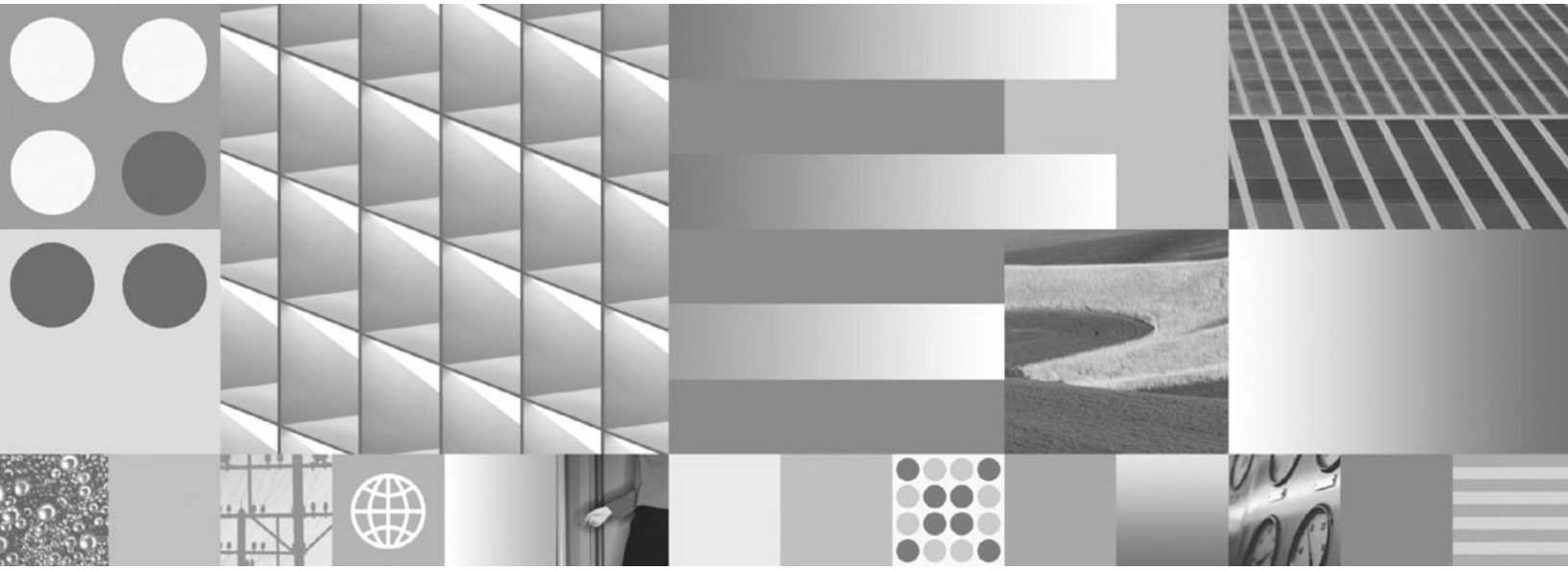
Version 5 Release 0



SDK Guide

IBM SDK for z/OS platforms, Java Technology Edition

Version 5 Release 0



SDK Guide

Note
Before using this information and the product it supports, read the information in "Notices" on page 81.

Contents

Preface v

Chapter 1. Overview 1
Version compatibility 1
Migrating from other IBM JVMs 1
Supported hardware 2
The z/OS batch toolkit 2

Chapter 2. Contents of the SDK and Runtime Environment 5
Contents of the Runtime Environment 5
Contents of the SDK 6

Chapter 3. Installing and configuring the SDK 9
Working with BPXPRM settings 9
Setting the region size 10
Setting MEMLIMIT 10
Setting LE runtime options 10
Setting LE 31-bit runtime options 11
Setting LE 64-bit runtime options 12
Marking failures 12
Setting the path 12
Setting the class path 13

Chapter 4. Running Java applications 15
The java and javaw commands 15
Obtaining version information 15
Specifying Java options and system properties . . 16
Standard options 16
Globalization of the java command 17
The Just-In-Time (JIT) compiler 18
Disabling the JIT 18
Enabling the JIT 18
Determining whether the JIT is enabled 19
Specifying garbage collection policy 19
Garbage collection options 20
Pause time 20
Pause time reduction 20
Environments with very full heaps 21
Euro symbol support 21
Support for Serbian locale 22

Chapter 5. Developing Java applications 23
Transforming XML documents 23
Using an older version of Xerces or Xalan . . . 24
Debugging Java applications. 25
Java Debugger (JDB) 25
Determining whether your application is running on a 31-bit or 64-bit JVM 26
How the JVM processes signals. 26
Signals used by the JVM 27
Linking a native code driver to the signal-chaining library 28
Writing JNI applications 29
Native formatting of Java types long, double, float 30
Support for thread-level recovery of blocked connectors 30
CORBA support 31
System properties for tracing the ORB 32
System properties for tuning the ORB 33
Java security permissions for the ORB 33
ORB implementation classes 34
RMI over IIOP 34
Implementing the Connection Handler Pool for RMI 35
Enhanced BigDecimal 35
Working in a multiple network stack environment 35
Using IBMJCECCA 36
Support for XToolkit 37
Support for the Java Attach API 37

Chapter 6. Applet Viewer 41
Distributing Java applications 41

Chapter 7. Class data sharing between JVMs 43
Overview of class data sharing 43
Class data sharing command-line options 44
Creating, populating, monitoring, and deleting a cache 46
Performance and memory consumption 47
Considerations and limitations of using class data sharing 47
Cache size limits. 47
Required APAR for Shared Classes 48
Working with BPXPRMxx settings 48
Runtime bytecode modification 48
Operating system limitations 49
Using SharedClassPermission 49
Adapting custom classloaders to share classes . . . 50

Chapter 8. Service and support for independent software vendors 51

Chapter 9. Accessibility 53
Keyboard traversal of JComboBox components in Swing 53

Chapter 10. Any comments on this user guide? 55

Appendix A. Command-line options . . 57
Specifying command-line options 57
General command-line options 58
System property command-line options 59

JVM command-line options 61
-XX command-line options 68
JIT command-line options 68
Garbage Collector command-line options 69

Appendix B. Known limitations 77

Notices 81
Trademarks 82

Preface

This user guide provides general information about the IBM® 64-bit SDK for z/OS®, Java™ 2 Technology Edition, Version 5.0. The user guide gives specific information about any differences in the IBM implementation compared with the Sun implementation.

Read this user guide with the more extensive documentation on the Sun Web site: http://java.sun.com.

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.

Technical changes made for this version of the user guide, other than minor or obvious ones, are indicated by blue chevrons when viewing in an Information Center, by blue chevrons and in red when viewing in HTML, or by vertical bars to the left of the changes when viewing as a PDF file.

The Program Code is not designed or intended for use in real-time applications such as (but not limited to) the online control of aircraft, air traffic, aircraft navigation, or aircraft communications; or in the design, construction, operation, or maintenance of any nuclear facility.

Chapter 1. Overview

The IBM SDK is a development environment for writing and running applets and applications that conform to the Java 5.0 Core Application Program Interface (API).

Version compatibility

In general, any application that ran with a previous version of the SDK should run correctly with the IBM 64-bit SDK for z/OS, v5.0. Classes compiled with this release are not guaranteed to work on previous releases.

For information about compatibility issues between releases, 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

If you are using the SDK as part of another product (for example, IBM WebSphere® Application Server), and you upgrade from a previous level of the SDK, perhaps v1.4.2, serialized classes might not be compatible. However, classes are compatible between service refreshes.

Migrating from other IBM JVMs

From Version 1.4.2 (64-bit) and Version 5 (31-bit), the IBM Runtime Environment for z/OS contains a new version 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 "JVM command-line options" on page 61.
- 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 produced have changed. You must change any

programs that process the verbose GC data to make them 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 included JVM specific classes in a file called core.jar. From Java 5.0 onwards, these are included in a file called `vm.jar`.
- Earlier versions of the IBM JRE included a file called `rt.jar` in the `jre/lib` directory. From Java v1.4 onwards, this file has been replaced by multiple JAR files in the `jre/lib` directory.
- For additional industry compatibility information, see Sun's Java 5 Compatibility Documentation: http://java.sun.com/j2se/5.0/compatibility.html
- For additional deprecated API information, see Sun's Java 5 Deprecated API List: http://java.sun.com/j2se/1.5.0/docs/api/deprecated-list.html
- All z/OS Java SDK program products can be installed and executed on the same z/OS system. They are independent program products and can coexist in any combination.
- The serial reusability feature of the IBM SDK for z/OS, version 1.4.2 (31-bit) and earlier, started using **-Xresettable**, is not supported. If you specify **-Xresettable** the JVM will issue an error message and will not start. The **-Xinitacsh** and **-Xinitth** options, which allowed heap sizes to be specified for the resettable JVM, are ignored. You can share data between JVMs in an address space (the old **-Xjvmset** and **-Xscmax** options) using Chapter 7, "Class data sharing between JVMs," on page 43, a new facility for 5.0. If you specify **-Xjvmset** or **-Xscmax** the JVM will issue an error message and will not start.
- The system property **os.arch** for IBM SDK for z/OS, version 1.4.2 (31-bit) versions and earlier had a value of **390**. From Java 5.0 onwards, the value of **os.arch** is **s390**.
- Tracing class dependencies, started using **-verbose:Xclassdep**, is not supported. If you specify **-verbose:Xclassdep**, the JVM will issue an error message and will not start.
- The JVM detects the operating system locale and sets the language preferences accordingly. For example, if the locale is set to **fr_FR**, JVM messages will be printed in French. To avoid seeing JVM messages in the language of the detected locale, remove the file `$SDK/jre/bin/java_xx.properties` where *xx* is the locale, such as **fr**, and the JVM will print messages in English.

Supported hardware

The z/OS 31-bit and 64-bit SDKs run on System z9® and zSeries® hardware.

The SDKs run on the following servers or equivalents:

- z9-109
- z990
- z900
- z890
- z800

The z/OS batch toolkit

From Version 5, Service Refresh 3 onwards, the z/OS products have been enhanced with the JZOS batch toolkit. This toolkit addresses many of the functional and environmental shortcomings in the previous Java batch capabilities on z/OS. The enhancements include a native launcher for running Java applications directly as

batch jobs or started tasks and a set of Java methods that make access to traditional z/OS data and key system services directly available from Java applications. Additional system services include console communication, multiline WTO (write to operator), and return code passing capability. For more details, see http://www.ibm.com/servers/eserver/zseries/software/java/jzos/overview.html and http://www.ibm.com/servers/eserver/zseries/software/java/.

Chapter 2. 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 must 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) might result in application portability problems.

The user guides, Javadoc files, and the accompanying copyright files are the only documentation included in this SDK for z/OS. 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. Additional z/OS related information is available on the z/OS Java Web site at http://www.ibm.com/servers/eserver/zseries/software/java/.

Contents of the Runtime Environment

A list of classes, tools, and other files that you can use with the standard Runtime Environment.

- Core Classes - These classes are the compiled class files for the platform and must remain compressed for the compiler and interpreter to access them. Do not modify these classes; instead, create subclasses and override where you need to.
- Trusted root certificates from certificate signing authorities - These certificates are used to validate the identity of signed material.
- JRE tools - The following tools are part of the Runtime Environment and are in the /usr/lpp/java/J5.0[_64]/jre/bin directory unless otherwise specified.

  ikeyman (iKeyman GUI utility)
  Allows you to manage keys, certificates, and certificate requests. For more information see the accompanying Security Guide and http://public.dhe.ibm.com/software/dw/jdk/security/50/GSK7c_SSL_IKM_Guide.pdf. The SDK also provides a command-line version of this utility.

  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.

  jextract (Dump extractor)
  Converts a system-produced dump into a common format that can be used by jdmpview. For more information, see jdmpview.

  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).

rmiregistry (Java remote object registry)
Creates and starts a remote object registry on the specified port of the current host.

tnameserv (Common Object Request Broker Architecture (CORBA) transient naming service)
Starts the CORBA transient naming service.

Contents of the SDK

A list of tools and reference information that is included with the standard SDK.

The following tools are part of the SDK and are located in the /usr/lpp/java/J5.0[_64]/bin directory:

appletviewer (Java Applet Viewer)
Tests and runs applets outside a Web browser.

apt (Annotation Processing Tool)
Finds and executes annotation processors based on the annotations present in the set of specified source files being examined.

extcheck (Extcheck utility)
Detects version conflicts between a target jar file and currently-installed extension jar files.

idlj (IDL to Java Compiler)
Generates Java bindings from a given IDL file.

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.

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.

javac (Java Compiler)
Compiles programs that are written in the Java programming language into bytecodes (compiled Java code).

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.

javap (Class File Disassembler)
Disassembles compiled files and can print a representation of the bytecodes.

jconsole (JConsole Monitoring and Management Tool)
Experimental (unsupported). Monitors local and remote JVMs using a GUI. JMX-compliant.

jdmpview (Cross-platform dump formatter)
Analyzes dumps. For more information, see the Diagnostics Guide.

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.

serialver (Serial Version Command)
Returns the serialVersionUID for one or more classes in a format that is suitable for copying into an evolving class.

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
The copyright notice for the SDK for z/OS software.

fixes.lst
A text file that describes any defects that are fixed after the initial release of this version.

Chapter 3. Installing and configuring the SDK

See the z/OS Web site for instructions about ordering, downloading, installing, and verifying the SDK.

http://www.ibm.com/servers/eserver/zseries/software/java/

Working with BPXPRM settings

Some of the parameters in PARMLIB member **BPXPRMxx** might affect successful Java operation by imposing limits on resources that are required.

The parameters described here do not cover those required for Class data sharing. See "Considerations and limitations of using class data sharing" on page 47 for the parameters required for Class data sharing.

Enter the z/OS operator command `D OMVS,O` to display the current **BPXPRMxx** settings. Enter the command `D OMVS,L` to show the highwater usage for some of the limits. If you configure the **BPXPRMxx LIMMSG** parameter to activate the support, **BPXInnnI** messages are displayed when the usage approaches and reaches the limits. You can use the `SETOMVS` command to change the settings without requiring an IPL.

Other products might impose their own requirements, but for Java the important parameters and their suggested minimum values are:

Table 1. BPXPRM settings

Parameter	Value
MAXPROCSYS	900
MAXPROCUSER	512
MAXUIDS	500
MAXTHREADS	10 000
MAXTHREADTASKS	5 000
MAXASSIZE	2 147 483 647
MAXCPUTIME	2 147 483 647
MAXMMAPAREA	40 960
IPCSEMNIDS	500
IPCSEMNSEMS	1 000
SHRLIBRGNSIZE	67 108 864
SHRLIBMAXPAGES	4 096

The lower of **MAXTHREADS** and **MAXTHREADTASKS** limits the number of threads that can be created by a Java process.

MAXMMAPAREA limits the number of 4K pages that are available for memory-mapped jar files through the environment variable **JAVA_MMAP_MAXSIZE**.

SHRLIBRGNSIZE controls how much storage is reserved in each address space for mapping shared DLLs that have the +l extended attribute set. If this storage space is exceeded, DLLs are loaded into the address space instead of using a single copy of USS storage that is shared between the address spaces. Some of the Java SDK DLLs have the +l extended attribute set. The z/OS command D OMVS,L shows the **SHRLIBRGNSIZE** size and peak usage. If this size is set to a much higher value than is needed, Java might have problems acquiring native (z/OS 31-bit) storage, which can cause a z/OS abend, such as 878-10, or a Java OutOfMemoryError.

SHRLIBMAXPAGES is only available in z/OS 1.7 and earlier releases. This parameter is similar to **SHRLIBRGNSIZE** except that it is a number of 4K pages and only applies to DLLs that have the .so suffix, but without the +l extended attribute. This feature requires Extended System Queue Area (ESQA), therefore you should use it carefully.

For further information about the use of these parameters, refer to the *z/OS MVS™ Initialization and Tuning Reference (SA22-7592)* at http://publibz.boulder.ibm.com/epubs/pdf/iea2e280.pdf and the *z/OS Unix System Services Planning Guide (GA22-7800)* at http://publibz.boulder.ibm.com/epubs/pdf/bpxzb280.pdf.

Setting the region size

Java requires a suitable z/OS region size to operate successfully. It is suggested that you do not restrict the region size, but allow Java to use what is necessary. Restricting the region size might cause failures with storage-related error messages or abends such as 878-10.

The region size might be affected by the following factors:

- **JCL REGION** parameter
- **BPXPRMxx MAXASSIZE** parameter
- RACF OMVS segment **ASSIZEMAX** parameter
- IEFUSI

You might want to exclude OMVS from using the IEFUSI exit by setting **SUBSYS(OMVS,NOEXITS)** in PARMLIB member SMFPRMxx.

For further information, see the documentation about the host product under which Java runs.

Setting MEMLIMIT

z/OS uses region sizes to determine the amount of storage available to running programs. For the 64-bit product, set the **MEMLIMIT** parameter to include at least 1024 MB plus the largest expected JVM heap size value **-Xmx**.

See *Limiting Storage use above the bar in z/Architecture* for information about setting the **MEMLIMIT** parameter: http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/FLASH10165.

Setting LE runtime options

LE runtime options can affect both performance and storage usage. The optimum settings will vary according to the host product and the Java application itself, but it is important to have good general settings.

The LE run-time options are documented in *Language Environment Programming Reference (SA22-7562)* at http://publibz.boulder.ibm.com/epubs/pdf/ceea3180.pdf.

Java and other products that are written in C or C++ might have LE runtime options embedded in the main programs by using `#pragma runopts`. These options are chosen to provide suitable default values that assist the performance in a typical environment. Any runtime overrides that you set might alter these values in a way that degrades the performance of Java or the host product. The host product's documentation might provide details of the product's default settings. Changes to the product's `#pragma runopts` might occur as a result of version or release changes. For details of how LE chooses the order of precedence of its runtime options, refer to the *Language Environment Programming Guide (SA22-7561)* at http://publibz.boulder.ibm.com/epubs/pdf/ceea2180.pdf.

Use the LE runtime option `RPTOPTS(ON)` as an override to write the options that are in effect, to stderr on termination. See the host product documentation and the *Language Environment Programming Guide (SA22-7561)* at http://publibz.boulder.ibm.com/epubs/pdf/ceea2180.pdf for details of how to supply LE runtime overrides. Before creating runtime overrides, run the application without overrides, to determine the existing options based on LE defaults and `#pragma` settings.

To tune the options, use the LE runtime option `RPTSTG(ON)` as an override, but be aware that performance could be reduced when you use this option. The output for `RPTSTG(ON)` also goes to stderr on termination. The *Language Environment Debugging Guide (GA22-7560)* at http://publibz.boulder.ibm.com/epubs/pdf/ceea1180.pdf explains `RPTSTG(ON)` output.

Setting LE 31-bit runtime options

There are a number of LE 31-bit options that are important for successful Java operation.

The following are the important options:

- **ANYHEAP**
- **HEAP**
- **HEAPPOOLS**
- **STACK**
- **STORAGE**
- **THREADSTACK**

You can change any, or all, of these options, however if you set the wrong values this might affect performance. The following values are a suggested starting point for these options:

```
ANYHEAP(2M,1M,ANY,FREE)
HEAP(80M,4M,ANY,KEEP)
HEAPPOOLS(ON,8,10,32,10,128,10,256,10,1024,10,2048,10,0,10,0, 10,0,10,0,10,0,10,0,10)
STACK(64K,16K,ANY,KEEP,128K,128K)
STORAGE(NONE,NONE,NONE,0K)
THREADSTACK(OFF,64K,16K,ANY,KEEP,128K,128K)
```

ANYHEAP and **HEAP** initial allocations (parameter 1) might be too large for transaction-based systems such as CICS®. Java applications that use many hundreds of threads might need to adjust the **STACK** initial and increment allocations (parameters 1, 2, 5 and 6) based on the **RPTSTG(ON)** output, which shows the maximum stack sizes that are used by a thread inside the application.

HEAPPOOLS(ON) should improve performance, but the LE-supplied default settings for the cell size and percentage pairs are not optimized for the best performance or storage usage.

For additional information, including how to set the LE runtime options, see:

- the Diagnostics Guide
- the *z/OS Language Environment Programming Reference (SA22-7562)* at http://publibz.boulder.ibm.com/epubs/pdf/ceea3180.pdf
- the *z/OS Language Environment Programming Guide (SA22-7561)* at http://publibz.boulder.ibm.com/epubs/pdf/ceea2180.pdf
- the host product documentation

Setting LE 64-bit runtime options

There are 64-bit versions of some of the runtime options.

The following are the 64-bit options:

- **HEAP64**
- **HEAPPOOLS64**
- **STACK64**
- **THREADSTACK64**

A suggested start point for HEAP64 as an override is `HEAP64(512M,4M,KEEP,16M,4M,KEEP,0K,0K,FREE)`.

The following are LE defaults, and should be appropriate:

```
STACK64(1M,1M,128M)
THREADSTACK64(OFF,1M,1M,128M)
HEAPPOOLS64(OFF,8,4000,32,2000,128,700,256,350.1024,100,2048,50,3072,50,4096,50,8192,25,16384,10,32768,5,65536,5)
```

Before you set an override for HEAPPOOLS64, use **RPTOPTS(ON)** or **RPTSTG(ON)** and check the result of `#pragma runopts`. Check this because the host product might have already set cell sizes and numbers that are known to produce good performance.

Also, these settings are dependant on a suitable **MEMLIMIT** setting. Based on these suggested LE 64-bit runtime options, the JVM requirement is a minimum of 512 MB as set for **HEAP64** (which should include **HEAPPOOLS64**), plus an initial value for **STACK64** of 1 MB times the expected maximum number of concurrent threads, plus the largest expected JVM heap **–Xmx** value.

Marking failures

The Java launcher can mark the z/OS Task Control Block (TCB) with an abend code when the launcher fails to load the VM or is terminated by an uncaught exception. To start TCB marking, set the environment variable **IBM_JAVA_ABEND_ON_FAILURE=Y**.

By default, the Java launcher will not mark the TCB.

Setting the path

If you alter the **PATH** environment variable, you will override any existing Java launchers in your path.

About this task

The **PATH** environment variable enables z/OS to find programs and utilities, such as javac, java, and javadoc tool, from any current directory. To display the current value of your **PATH**, type the following at a command prompt:

```
echo $PATH
```

To add the Java launchers to your path:

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

   ```
   export PATH=/usr/lpp/java/J5.0[_64]/bin:/usr/lpp/java/J5.0[_64]/jre/bin:$PATH
   ```

2. Log on again or run the updated shell script to activate the new **PATH** environment variable.

Results

After setting the path, you can run a tool by typing its name at a command prompt from any directory. For example, to compile the file `Myfile.Java`, at a command prompt, type:

```
javac Myfile.Java
```

Setting the class path

The class path tells the SDK tools, such as java, javac, and the javadoc tool, where to find the Java class libraries.

About this task

You should set the class path explicitly only if:

- 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 command 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.

Chapter 4. Running Java applications

Java applications can be started using the java launcher or through JNI. Settings are passed to a Java application using command-line arguments, environment variables, and properties files.

The java and javaw commands

An overview of the java and javaw commands.

Purpose

The java and javaw tools start a Java application by starting a Java Runtime Environment and loading a specified class.

The javaw command is identical to java, and is supported on z/OS for compatibility with other platforms.

Usage

The JVM searches for the initial class (and other classes that are used) in three sets of locations: the bootstrap class path, the installed extensions, and the user class path. 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 start. 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 arguments to be passed to the main() function of the startup class.

Obtaining version information

You obtain The IBM build and version number for your Java installation using the **-version** option.

1. Open a shell prompt.
2. Type the following command:

   ```
   java -version
   ```

You will see information similar to:

```
java version "1.5.0"
Java(TM) 2 Runtime Environment, Standard Edition (build pxi32dev-20051104)
IBM J9 VM (build 2.3, J2RE 1.5.0 IBM J9 2.3 Linux x86-32 j9vmxi3223-20051103 (JIT enabled)
J9VM - 20051027_03723_lHdSMR
JIT  - 20051027_1437_r8
GC   - 20051020_AA)
JCL  - 20051102
```

Exact build dates and versions will change.

Specifying Java options and system properties

You can specify Java options and system properties on the command line, by using an options file, or by using an environment variable.

About this task

These methods of specifying Java options are listed in order of precedence. Rightmost options on the command line have precedence over leftmost options; for example, if you specify:

```
java -Xint -Xjit myClass
```

The **-Xjit** option takes precedence.

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=**<*file*>.
3. By creating an environment variable called **IBM_JAVA_OPTIONS** containing the options. For example:
   ```
   export IBM_JAVA_OPTIONS="-Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump"
   ```

Standard options

The definitions for the standard options.

See "JVM command-line options" on page 61 for information about nonstandard (-X) options.

-agentlib:<*libname*>**[=**<*options*>**]**
Loads a 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 a native agent library by full path name.

-assert
Prints help on assert-related options.

-cp <*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 class path is, by default, the current directory (.).

-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 class path is, by default, the current directory (.).

-D<*property name*>**=**<*value*>
Sets a system property.

-help or -?
Prints a usage message.

-javaagent:<*jarpath*>**[=**<*options*>**]**
Load a 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*>**[,**<*option*>**...]**
Enables verbose output. Separate multiple options using commas. The available options are:

class
Writes an entry to stderr for each class that is loaded.

gc Writes verbose garbage collection information to stderr. Use **-Xverbosegclog** (see "Garbage Collector command-line options" on page 69 for more information) to control the output. See the Diagnostics Guide for more information.

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 product version.

-version:<*value*>
Requires the specified version to run, for example "1.5".

-X Prints help on nonstandard options.

Globalization of the java command

The java and javaw launchers accept arguments and class names containing 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 to 9, A to 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 provide translated 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 is in English.

The Just-In-Time (JIT) compiler

The IBM Just-In-Time (JIT) compiler dynamically generates machine code for frequently used bytecode sequences in Java applications and applets during their execution. The JIT v5.0 compiler delivers new optimizations as a result of compiler research, improves optimizations implemented in previous versions of the JIT, and provides better hardware exploitation.

The JIT is included in both the IBM SDK and Runtime Environment, which is enabled by default in user applications and SDK tools. Typically, you do not start the JIT explicitly; the compilation of Java bytecode to machine code occurs transparently. You can disable the JIT to help isolate a problem. If a problem occurs when executing a Java application or an applet, you can disable the JIT to help isolate the problem. Disabling the JIT is a temporary measure only; the JIT is required to optimize performance.

For more information about the JIT, see the Diagnostics Guide.

Disabling the JIT

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

About this task

Turning off the JIT is a temporary measure that can help isolate problems when debugging Java applications.

- Set the **JAVA_COMPILER** environment variable to `NONE` or the empty string before running the java application. Type the following at a shell prompt:

  ```
  export 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 <class>
  ```

- Use the **-Xint** option on the JVM command line. Type the following at a shell prompt:

  ```
  java -Xint <class>
  ```

Enabling the JIT

The JIT is enabled by default. You can explicitly enable the JIT 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:

  ```
  export JAVA_COMPILER=jitc
  ```

  If the **JAVA_COMPILER** environment variable is an empty string, the JIT remains disabled. To disable the environment variable, at the prompt, enter:

  ```
  unset JAVA_COMPILER
  ```
- Use the **-D** option on the JVM command line to set the **java.compiler** property to `jitc`. At a prompt, enter:

  ```
  java -Djava.compiler=jitc <class>
  ```
- Use the **-Xjit** option on the JVM command line. Do **not** specify the **-Xint** option at the same time. At a prompt, enter:

  ```
  java -Xjit <class>
  ```

Determining whether the JIT is enabled

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

Run the java launcher with 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)
```

What to do next

For more information about the JIT, see the Diagnostics Guide.

Specifying garbage collection policy

The Garbage Collector manages the memory used by Java and by applications running in the JVM.

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 that 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 about garbage collection, see the Diagnostics Guide.

Garbage collection options

The **-Xgcpolicy** options control the behavior of the Garbage Collector. They make trade-offs between throughput of the application and overall system, and the pause times that are caused by garbage collection.

The format of the option and its values is:

-Xgcpolicy:optthruput
(Default and recommended value.) Delivers very high throughput to applications, but at the cost of occasional pauses.

-Xgcpolicy:optavgpause
Reduces the time spent in 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.

-Xgcpolicy:gencon
Requests the combined use of concurrent and generational GC to help minimize the time that is spent in any garbage collection pause.

-Xgcpolicy:subpool
Uses an improved object allocation algorithm to achieve better performance when allocating objects on the heap. This option might improve performance on large SMP systems.

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 garbage collection 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 "new" and the "tenure" areas. Objects are placed in one of these areas depending on their age. The new area is the smaller of the two and contains new objects; the tenure is larger and contains older objects. Objects are first allocated to the new area; if they have active references for long enough, they are promoted to the tenure area.

Generational GC depends on most objects not lasting long. Generational GC reduces pause times by concentrating the effort to reclaim storage on the new area because it has the most recyclable space. Rather than occasional but lengthy pause times to collect the entire heap, the new area is collected more frequently and, if the new area is small enough, pause times are comparatively short. However, generational GC has the drawback that, over time, the tenure area might become full. 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 garbage collection options are used; and, if requests for more heap space continue to be made, the application might receive an OutOfMemoryError, which results in JVM termination if the exception is not caught and handled. At this point, the JVM produces a Javadump file for use during diagnostics. 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.

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. From 1 January 2008, Cyprus and Malta also have the Euro as the default currency.

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.

Support for Serbian locale

From Service Refresh 5, the following new locale is added: Serbia (SE), with three new locale variations.

The locale variations are:

- sr_RS
- sr_Cyrl_RS
- sr_Latn_RS

The existing locale variations for the former Serbia and Montenegro are maintained as before. The 3-letter country code SRB, corresponding to the 2-letter country code RC, is also added.

Chapter 5. Developing Java applications

The SDK for z/OS contains many tools and libraries required for Java software development.

See “Contents of the SDK” on page 6 for details of the tools available.

Transforming XML documents

The IBM SDK contains the XSLT4J processor and the XML4J parser. With these tools, you can 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.

About this task

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.

With the XSLT4J processor, you choose between the original XSLT Interpretive processor and the XSLT Compiling processor. The Interpretive processor is 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 for high performance runtime environments; it generates a transformation engine, or *translet*, from an XSL style sheet. This approach separates the interpretation of stylesheet instructions from their runtime application to XML data.

The XSLT Interpretive processor is the default processor. To use the XSLT Compiling processor:

- Change the entry in the `jaxp.properties` file, (located in /usr/lpp/java/J5.0[_64]/jre/lib) or
- Set the **javax.xml.transform.TransformerFactory** system property to `org.apache.xalan.xsltc.trax.TransformerFactoryImpl`.

To implement properties in the `jaxp.properties` file, copy `jaxp.properties.sample` to `jaxp.properties` in /usr/lpp/java/J5.0[_64]/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 /usr/lpp/java/J5.0[_64]/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 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 (before 2.0) or Xalan (before 2.3) in the endorsed override, you might get a NullPointerException when you start your application. This exception occurs because these older versions do not handle the `jaxp.properties` file correctly.

About this task

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 (https://jaxp.dev.java.net/).
- Remove the `jaxp.properties` file from /usr/lpp/java/J5.0[_64]/jre/lib.
- Uncomment the entries in the `jaxp.properties` file in /usr/lpp/java/J5.0[_64]/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** in your application. 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.

```
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
```

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 the operating system.

More information about problem diagnosis using Java can be found in the Diagnostics Guide.

Java Debugger (JDB)

The Java Debugger (JDB) is included in the SDK for z/OS. The debugger is started with 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>
   ```

   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 start.

For more information about JDB options, type:

```
jdb -help
```

For more information 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 workstations. 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 JVM:

   ```
   jdb -attach <host>:<port>
   ```

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 about 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 31-bit or 64-bit JVM

Some Java applications must be able to determine whether they are running on a 31-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 31- and 64-bit forms for platforms that support both 31- 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 31- and 64-bit code.

About this task

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 31-bit mode
- 64 - the JVM is running in 64-bit mode

You can inspect the **com.ibm.vm.bitmode** property from inside your application code using the call:

```
System.getProperty("com.ibm.vm.bitmode");
```

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. Produces dumps, 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 unexpected-shutdown hook
  4. Performs the necessary JVM cleanup

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:

1. Calls your application's signal handler for that signal
2. Calls all application shutdown hooks

3. Calls any application-installed exit hook
4. 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 stop. 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 2 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 an unrecoverable 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 shut down.

Controls
Other signals that are used by the JVM for control purposes.

Table 2. 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 call an unknown machine instruction)	Yes
SIGFPE (8)	Exception	Floating point exception (divide by zero)	Yes
SIGABRT (6)	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

Table 2. Signals used by the JVM (continued)

Signal Name	Signal type	Description	Disabled by -Xrs
SIGQUIT (3)	Control	By default, this triggers a Javadump.	Yes
SIGRECONFIG (58)	Control	Reserved to detect any change in the number of CPUs, processing capacity, or physical memory.	Yes
SIGTRAP (5)	Control	Used by the JIT.	Yes
SIGCHLD (17)	Control	Used by the SDK for internal control.	No
SIGUSR1	Control	Used by the SDK.	No

Note: A number supplied after the signal name is the standard numeric value for that signal.

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.

About this task

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.

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.

The environment variable **JAVA_HOME** should be set to the location of the SDK, for example,/usr/lpp/java/J5.0[_64]/.

To use libjsig.a:

- Link it with the application that creates or embeds a JVM:

```
cc_r -q64 <other compile/link parameter> -L/usr/lpp/java/J5.0[_64]/jre/bin -ljsig
-L/usr/lpp/java/J5.0[_64]/jre/bin/j9vm -ljvm java_application.c
```

  Note: Use `xlc_r` or `xlC_r` in place of `cc_r` if that is how you usually call the compiler or linker.

Writing JNI applications

Valid Java Native Interface (JNI) version numbers that 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 JNI is not supported.

This version number determines only the level of the JNI to use. The actual level of the JVM that is created is specified by the JSE libraries (that is, v5.0). The JNI level *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 31- 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.

For more information about writing 64-bit applications, see the IBM Redpaper *z/OS 64-bit C/C++ and Java Programming Environment* at http://www.redbooks.ibm.com/abstracts/redp9110.html.

ASCII and EBCDIC issues

On z/OS, the Java Virtual Machine is essentially an EBCDIC application. Enhanced ASCII methods are C or C++ code that has been compiled with ASCII compiler options. If you create JNI routines as enhanced ASCII C or C++ methods you will be operating in a bimodal environment; your application will be crossing over between ASCII and EBCDIC environments.

The inherent problem with bimodal programs is that, in the z/OS runtime, threads are designated as either EBCDIC or enhanced ASCII and are not intended to be switched between these modes in typical use. Enhanced ASCII is not designed to handle bimodal issues. You might get unexpected results or experience failures when the active mode does not match that of the compiled code. There are z/OS runtime calls that applications might use to switch the active mode between EBCDIC and enhanced ASCII (the __ae_thread_swapmode() and __ae_thread_setmode() functions are documented in Language Environment® Vendor Interfaces, see the *SA22-7568-06 Red Book*: http://publibz.boulder.ibm.com/epubs/pdf/ceev1160.pdf). However, even if an application is carefully coded to switch modes correctly, other bimodal issues might exist.

Native formatting of Java types long, double, float

The latest C/C++ compilers and runtimes can convert jlong, jdouble, and jfloat data types to strings by using printf()-type functions.

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 no longer supported by Version 6 of the SDK for z/OS. 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:

- /usr/lpp/java/J5.0[_64]/demo/jni/JNINativeTypes/c/convert.o (For 31-bit)
- /usr/lpp/java/J5.0[_64]/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))
```

A C/C++ application that returns a jfloat data type to a Java application must be compiled with the FLOAT (IEEE) C/C++ compiler option. Applications compiled without this option will return incorrect data types. Further information about compiling C/C++ source code, which applies to this Java release, can be found in the support document http://www-03.ibm.com/servers/eserver/zseries/software/java/usingjni.html#building

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 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 information for these classes is provided with the SDK in the `docs/apidoc.zip` file.

CORBA support

The Java 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: http://java.sun.com/j2se/1.5.0/docs/api/org/omg/CORBA/doc-files/compliance.html.

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 *formal/99-10-07*.

http://www.omg.org/cgi-bin/doc?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 that ORB services can use to 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 have 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=<*file*>
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 enable 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.

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. Do not set the timeout too low to avoid connections ending unnecessarily.

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 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, which might result in a SecurityException. If your program uses any of these methods, ensure that it is granted the necessary permissions.

Table 3. Methods affected when running with Java 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

Table 3. Methods affected when running with Java SecurityManager (continued)

Class/Interface	Method	Required permission
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
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) 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 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 Interface Description Language (IDL) program (demo/rmi-iiop/idl)
- The *Java Language to IDL Mapping* document is a detailed technical specification of RMI-IIOP: 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.

About this task

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.

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 is currently deprecated. 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.*;`.

Working in a multiple network stack environment

In a multiple network stack environment (CINET), when one of the stacks fails, no notification or Java exception occurs for a Java program that is listening on an INADDR_ANY socket. Also, when new stacks become available, the Java application does not become aware of them until it rebinds the INADDR socket.

To avoid this situation, when a TCP/IP stack comes online:

- If the **ibm.socketserver.recover** property is set to false (which is the default), an exception (NetworkRecycledException) is thrown to the application to allow it either to fail or to attempt to rebind.
- If the **ibm.socketserver.recover** property is set to true, Java attempts to redrive the socket connection on the new stack if listening on all addresses (addrs). If the socket bind cannot be replayed at that time, an exception (NetworkRecycledException) is thrown to the application to allow it either to fail or to attempt to rebind.

Both ServerSocket.accept() and ServerSocketChannel.accept() can throw NetworkRecycledException.

While a socket is listening for new connections, it maintains a queue of incoming connections. When NetworkRecycledException is thrown and the system attempts to rebind the socket, the connection queue is reset and connection requests in this queue are dropped.

Using IBMJCECCA

IBMJCECCA uses ICSF services during processing. You must have the correct CSFSERV access to use the ICSF services and IBMJCECCA.

Table 4. CSFSERV access permissions required to use ICSF services

ICSF APIs used by IBMJCECCA	CSF access required
CSNBSYE and **CSNESYE** (64-bit) Symmetric key encipher	**CSFENC** Encipher callable service **CSFCVE** Cryptographic variable encipher callable service
CSNBSYD and **CSNESYD** (64-bit) Symmetric key decipher	**CSFDEC** Decipher callable service
CSNBOWH and **CSNEOWH** (64-bit) One-way hash generate	**CSFOWH** One-way hash generate callable service
CSNBRNG and **CSNERNG** (64-bit) Random number generate	**CSFRNG** Random number generate callable service
CSNDKRC and **CSNFKRC** (64-bit) PKDS record create	**CSFPKRC** PKDS record create callable service **CSFKRC** Key record create callable service
CSNDKRD and **CSNFKRD** (64-bit) PKDS record delete	**CSFPKRD** PKDS record delete callable service **CSFKRD** Key record delete callable service
CSNDRKD and **CSNFRKD** (64-bit) Retained key delete	**CSFRKD** Retained key delete callable service
CSNDPKG and **CSNFPKG** (64-bit) PKA key generate	**CSFPKG** PKA key generate callable service
CSNDDSG and **CSNFDSG** (64-bit) Digital signature generate	**CSFDSG** Digital signature generate service
CSNDDSV and **CSNFDSV** (64-bit) Digital signature verify	**CSFDSV** Digital signature verify callable service
CSNDPKB and **CSNFPKB** (64-bit) PKA key token build	**CSFPKG** PKA key generate callable service **CSFPKTC** PKA key token change callable service
CSNDRKL and **CSNFRKL** (64-bit) Retained key list	**CSFRKL** Retained key list callable service
CSNDPKX and **CSNFPKX** (64-bit) PKA public key extract	**CSFPKX** PKA Public Key Extract callable service
CSNBENC and **CSNEENC** (64-bit) Encipher	**CSFENC** Encipher callable service
CSNBDEC and **CSNEDEC** (64-bit) Decipher	**CSFDEC** Decipher callable service
CSNDPKE and **CSNFPKE** (64-bit) PKA encrypt	**CSFPKE** PKA encrypt callable service
CSNDPKD and **CSNFPKD** (64-bit) PKA decrypt	**CSFPKD** PKA decrypt callable service
CSNDPKI and **CSNFPKI** (64-bit) PKA key import	**CSFPKI** PKA key import callable service

Table 4. CSFSERV access permissions required to use ICSF services (continued)

ICSF APIs used by IBMJCECCA	CSF access required
CSNBCKM and **CSNECKM** (64-bit) Multiple clear key import	**CSFCKM** Multiple clear key import callable service
CSNBKGN and **CSNEKGN** (64-bit) Key generate	**CSFKGN** Key generate callable service
CSNDSYI Symmetric key import	**CSFSYI** Symmetric key import callable service
CSNDSYX Symmetric key export	**CSFSYX** Symmetric key export callable service

Support for XToolkit

The IBM 64-bit SDK for z/OS, v5.0 supports XToolkit from Service Refresh 4. 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.

Restriction: Motif is no longer supported and will be removed in a later release.

Related links:

- An example, Integrating Swing into Eclipse RCPs: http://eclipsezone.com/eclipse/forums/t45697.html
- Reference Information in the Eclipse information center: http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/awt/SWT_AWT.html
- Set up information on the Sun Web site: http://java.sun.com/j2se/1.5.0/docs/guide/awt/1.5/xawt.html

Support for the Java Attach API

The Java Attach API allows your application to connect to another virtual machine (the "target"). Your application can then load an agent application into the target virtual machine, for example to perform tasks such as monitoring status.

Code for agent applications, such as JMX agents or JVMTI agents, is normally loaded during virtual machine startup by specifying special startup parameters. Requiring startup parameters might not be convenient for using agents on applications that are already running, such as WebSphere Application Servers. Using the Java Attach API, lets you load an agent at any time by specifying the process ID of the target virtual machine. The Attach API capability is sometimes called "late attach".

On z/OS, the Attach API is supported for Java 5 SR 10 and later. The Attach API is enabled by default on z/OS 31-bit for Java 5 SR 10 only. The Attach API is disabled by default on z/OS 64-bit, and on all supported z/OS platforms for Java 5 SR 11 and later.

Security considerations

Security for the Java Attach API is handled by UNIX® user and group file permissions. On z/OS, you must use UNIX user and group permissions to protect your applications. It is not sufficient to rely on RACF® or system level security to

protect your applications, because these mechanisms do not have the necessary UNIX user and group permissions set up and configured for the Java Attach API to remain secure.

The Java Attach API creates files and directories in a common directory. The common directory, subdirectories, and files in it, have UNIX file permissions. It is recommended that you change the ownership of the common directory to ROOT or another privileged user ID, to prevent 'spoofing' attacks.

The key security features of the Java Attach API are:

- For Java 5 SR 10, a process using the Java Attach API must belong to the same UNIX group as the target process. This ensures that only users in the same UNIX group can attach to another user's target process.
- For Java 5 SR 11 and later, a process using the Java Attach API must be owned by the same UNIX userid as the target process. This ensures that only the target process owner can attach other applications to the target process.
- For Java 5 SR 11 and later, access to the files or directories owned by a process is controlled by user permissions only; group access is disabled.
- The common directory uses the sticky bit to prevent a user from deleting or replacing another user's subdirectory. To preserve the security of this mechanism, set the ownership of the common directory to ROOT.
- The subdirectory for a process is accessible only by members of the same UNIX group as the owner of a process. For Java 5 SR 11 and later, access is restricted to the owner only.
- Information about the target process can be written only by the owner and read only by the owner or a member of the owner's group. For Java 5 SR 11 and later, access is restricted to the owner only.

You must secure access to the Java Attach API capability to ensure that only authorized users or processes can connect to another virtual machine. If you do not intend to use the Java Attach API capability, disable this feature using the Java system property. Do this by setting the **com.ibm.tools.attach.enable** system property to the value **no**; for example:

```
-Dcom.ibm.tools.attach.enable=no
```

Using the Java Attach API

By default, the target virtual machine is identified by its process ID. To use a different target, change the system property **com.ibm.tools.attach.id**; for example:

```
-Dcom.ibm.tools.attach.id=<process_ID>
```

The target process also has a human-readable "display name". By default, the display name is the process ID. To change the default display name, use the **com.ibm.tools.attach.displayName** system property. The ID and display name cannot be changed after the application has started.

The Attach API creates working files in a common directory called `.com_ibm_tools_attach`, which is created in the system temporary directory. The system property **java.io.tmpdir** holds the value of the system temporary directory. On non-Windows® systems, the system temporary directory is typically `/tmp`. To modify the working directory, use the Java system property **com.ibm.tools.attach.directory**; for example:

```
-Dcom.ibm.tools.attach.directory=/working
```

If your Java application ends abnormally, for example, following a crash or a SIGKILL signal, the process subdirectory is not deleted. The Java VM detects and removes obsolete subdirectories where possible. The subdirectory can also be deleted by the owning userid.

On heavily loaded system, applications might experience timeouts when attempting to connect to target applications. The default timeout is 120 seconds. Use the **com.ibm.tools.attach.timeout** system property to specify a different timeout value in seconds; for example, to timeout after 60 seconds:

```
-Dcom.ibm.tools.attach.timeout=60
```

A timeout value of zero indicates an indefinite wait.

For JMX applications, you might need to disable authentication by editing the `<JAVA_HOME>/jre/lib/management/management.properties` file. Set the following properties to disable authentication in JMX:

```
com.sun.management.jmxremote.authenticate=false
com.sun.management.jmxremote.ssl=false
```

An unsuccessful attempt to invoke the Attach API results in one of the following exceptions:

- `com.sun.tools.attach.AgentLoadException`
- `com.sun.tools.attach.AgentInitializationException`
- `java.io.IOException`

Related links:

- The Attach API: http://java.sun.com/javase/6/docs/technotes/guides/attach/index.html.

Chapter 6. Applet Viewer

The Java plug-in is used to run Java applications in the browser. The **appletviewer** is used to test applications designed to be run in a browser.

Distributing Java applications

Java applications typically consist of class, resource, and data files.

When you distribute a Java application, your software package probably consists of the following parts:
- Your own class, resource, and data files
- An installation procedure or program

Your SDK for z/OS software license does **not** allow you to redistribute any of the SDK's files with your application. You must ensure that a licensed version of the SDK for z/OS is installed on the target workstation.

When distributing your application for use on a z/OS platform, make the z/OS SDK a prerequisite, because z/OS does not have a separate JRE.

Chapter 7. Class data sharing between JVMs

Class data sharing allows multiple JVMs to share a single space in memory.

The Java Virtual Machine (JVM) allows you to share data between JVMs by storing it in a cache in shared memory. Sharing reduces the overall virtual storage 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 class data sharing

Class data sharing provides a transparent method of reducing memory footprint and improving JVM start-up time.

Enabling class data sharing

Enable class data sharing by using the **-Xshareclasses** option when starting a JVM. The JVM connects to an existing cache or creates a new cache if one does not exist.

All bootstrap and application classes loaded by the JVM are shared by default. Custom classloaders share classes automatically if they extend the application classloader; 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 50.

Cache access

Any JVM connected to a cache can update the cache. Any number of JVMs can concurrently read from the cache, even while another JVM is writing to it.

You must take care if runtime bytecode modification is being used. See “Runtime bytecode modification” on page 48 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.

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 classloader that has registered to share class data can update the shared class cache.

If a Java SecurityManager is installed, classloaders, excluding the default bootstrap, application, and extension classloaders, must be granted permission to share classes by adding SharedClassPermission lines to the `java.policy` file. See "Using SharedClassPermission" on page 49. The RuntimePermission createClassLoader restricts the creation of new classloaders and therefore also restricts access to the cache.

Cache lifespan

Multiple caches can exist on a system and you specify them by name as a suboption to the **-Xshareclasses** command. A JVM can connect to only one cache at any one time.

You can override the default cache size on startup using **-Xscmx<n><size>**. 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 "Class data sharing command-line options" or use **-Xshareclasses:help** to see a list of available suboptions.

Class data sharing command-line options

Class data sharing and the cache management utilities are controlled using command-line options to the Java launcher.

For options that take a <*size*> parameter, 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. The default cache size is platform-dependent. You can find out the size value being used by adding **-verbose:sizes** as a command-line argument. The minimum cache size is 4 KB. The maximum cache size is also platform-dependent. (See "Cache size limits" on page 47.)

-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 suboptions.

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 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 inserts 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 provides overall status on the shared class cache and more detailed error messages.

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 usual 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. Unrecoverable error messages, which prevent the JVM from initializing, are displayed.

nonfatal
Allows the JVM to start even if class data sharing fails. Normal behavior for the JVM is to refuse to start if class data sharing fails. If you select **nonfatal** and the shared classes cache fails to initialize, the JVM starts without class data sharing.

none
Can be added to the end of a command line to disable class data sharing. This suboption 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 suboption and a bytecode modification agent is installed, classes are 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 JVMs 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, all classes are stored twice in the cache. See "Runtime bytecode modification" on page 48 for more information.

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. This option is not a utility option because it does not cause the JVM to exit.

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 for the cache specified by the **name** suboption. Every class is listed in chronological order, 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

An overview of the life-cycle of a shared class data cache including examples of the cache management utilities.

To enable class data sharing, add **-Xshareclasses[:name=<name>]** to your application command line.

The JVM either connects to an existing cache of the given name or creates a new cache of that name. If a new cache is created, it is populated with all bootstrap and application classes being loaded until the cache becomes full. If two or more JVMs are started concurrently, they 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`. You can run these utilities 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, run `java -Xshareclasses:[name=<name>],destroy`. You usually delete caches only if they contain many stale classes or if the cache is full and you want to create a bigger cache.

You should tune the cache size for your specific application, because the default is unlikely to be the optimum size. To determine the optimum cache size, 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. 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 affect on the performance or capability of any JVMs connected to it, so it is acceptable to decide on a cache size that is smaller than required.

If a cache becomes full, a message is displayed on the command line of any JVMs using the **verbose** suboption. All JVMs sharing the full cache then loads 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 storage consumption. It is also useful on systems that frequently start up and shut down JVMs, which benefit from the improvement in startup time.

The processor and memory usage required 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 show greater overall startup time benefits.

Duplicate classes are consolidated in 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 storage consumption.

Considerations and limitations of using class data sharing

Consider these factors when deploying class data sharing in a product and using class data sharing in a development environment.

Cache size limits

The maximum theoretical cache size is 2 GB. The size of cache you can specify is limited by the amount of physical memory and swap space available to the system.

Because the virtual address space of a process is shared between the shared classes cache and the Java heap, if you increase the maximum size of the Java heap you might reduce the size of the shared classes cache you can create.

Required APAR for Shared Classes

You must apply z/OS APAR OA11519, available for z/OS R1.6 and onwards, to any z/OS system where shared classes are used. This APAR ensures that multiple shmat requests for the same shared segment will map to the same virtual address for multiple JVMs.

Without this APAR, there is a problem with using shared memory when multiple JVMs are stored in a single address space. Each shmat call consumes a separate virtual address range. This is not acceptable because shared classes will run out of shared memory pages prematurely.

Working with BPXPRMxx settings

Some of the **BPXPRMxx** parmlib settings affect shared classes performance. Using the wrong settings can stop shared classes from working. These settings might also have performance implications.

For further information about performance implications and use of these parameters, see the *z/OS MVS Initialization and Tuning Reference (SA22-7592)* at http://publibz.boulder.ibm.com/epubs/pdf/iea2e280.pdf and the *z/OS Unix System Services Planning Guide (GA22-7800)* at http://publibz.boulder.ibm.com/epubs/pdf/bpxzb280.pdf. The most significant **BPXPRMxx** parameters that affect the operation of shared classes are:

- **MAXSHAREPAGES**, **IPCSHMSPAGES**, **IPCSHMMPAGES**, and **IPCSHMMSEGS**. These settings affect the amount of shared memory pages available to the JVM. The JVM uses these memory pages for the shared classes cache. If you request large cache sizes, you might have to increase the amount of shared memory pages available.

  The shared page size for a z/OS Unix System Service is fixed at 4 KB for 31-bit and 1 MB for 64-bit. Shared classes try to create a 16 MB cache by default on both 31- and 64-bit platforms. Therefore set **IPCSHMMPAGES** greater than 4096 on a 31-bit system.

  If you set a cache size using **-Xscmx**, the VM will round up the value to the nearest megabyte. You must take this into account when setting **IPCSHMMPAGES** on your system.
- **IPCSEMNIDS**, and **IPCSEMNSEMS**. These settings affect the amount of SystemV IPC semaphore available to Unix processes. IBM shared classes use System V IPC semaphores to communicate between the JVMs.

Runtime bytecode modification

Any JVM using a JVM Tool Interface (JVMTI) agent that can modify bytecode data must use the **modified=**<*modified_context*> suboption if it wants to share the modified classes with another JVM.

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 JVM. 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 JVM, but with a small affect on performance. Using a modification context with a JVMTI agent avoids the need for extra checks and therefore has no affect 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. You can use the **modified=**<*modification_context*> suboption 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

Temporary disk space must be available to hold cache information. The operating system enforces cache permissions.

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 re-create 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 operating system enforces the permissions for accessing a shared class cache. If you do not specify a cache name, 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, you must grant these class loaders 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 the permissions of the default bootstrap, application, or extension class loaders.

Adapting custom classloaders to share classes

Any classloader that extends java.net.URLClassLoader can share classes without modification. You must adopt classloaders that do not extend java.net.URLClassLoader to share class data.

You must grant all custom classloaders shared class permissions if a SecurityManager is being used; see "Using SharedClassPermission" on page 49. IBM provides several Java interfaces for various types of custom classloaders, which allow the classloaders to find and store classes in the shared class cache. These classes are in the com.ibm.oti.shared package.

The Javadoc document for this package is provided with the SDK in the `docs/apidoc.zip` file.

See the Diagnostics Guide for more information about how to use these interfaces.

Chapter 8. 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 usual method of access or on the Web at: http://www.ibm.com/partnerworld/.

If you have purchased a service contract (that is, the IBM 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.

Chapter 9. Accessibility

The user guides that are supplied with this SDK and the Runtime Environment have been tested using screen readers.

To change the font sizes in the user guides, use the function that is supplied with your browser, typically 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/.

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 JComboBox does not change value until an item is selected. This is the correct behavior for this release and improves accessibility and usability by ensuring that the keyboard traversal behavior is consistent with mouse traversal behavior.

Chapter 10. Any comments on this user guide?

If you have any comments about this user guide, contact us through one of the following channels. Note that these channels are not set up to answer technical queries, but are for comments about the documentation only.

Send your comments:

- By e-mail to idrcf@hursley.ibm.com.
- By fax:
  - From the UK: 01962 842327
  - From elsewhere: +44 1962 842327
- By mail to:

  IBM United Kingdom Ltd
  User Technologies,
  Mail Point 095
  Hursley Park
  Winchester
  Hampshire
  SO21 2JN
  United Kingdom

The fine print. By choosing to send a message to IBM, you acknowledge that all information contained in your message, including feedback data, such as questions, comments, suggestions, or the like, shall be deemed to be non-confidential and IBM shall have no obligation of any kind with respect to such information and shall be free to reproduce, use, disclose, and distribute the information to others without limitation. Further, IBM shall be free to use any ideas, concepts, know-how or techniques contained in such information for any purpose whatsoever, including, but not limited to, developing, manufacturing and marketing products incorporating such information.

Appendix A. Command-line options

You can specify the options on the command line while you are starting Java. They override any relevant environment variables. For example, using **-cp <dir1>** with the Java command completely overrides setting the environment variable **CLASSPATH=<dir2>**.

This chapter provides the following information:

- "Specifying command-line options"
- "General command-line options" on page 58
- "System property command-line options" on page 59
- Nonstandard command-line options
- JIT command-line options
- Garbage Collector command-line options

Specifying command-line options

Although the command line is the traditional way to specify command-line options, you can pass options to the JVM in other ways.

Use only single or double quotation marks for command-line options when explicitly directed to do so for the option in question. Single and double quotation marks have different meanings on different platforms, operating systems, and shells. Do not use **'-X<option>'** or **"-X<option>"**. Instead, you must use **-X<option>**. For example, do not use `'-Xmx500m'` and `"-Xmx500m"`. Write this option as `-Xmx500m`.

These precedence rules (in descending order) apply to specifying options:

1. Command line.

   For example, `java -X<option> MyClass`
2. A file containing a list of options, specified using the **–Xoptionsfile** option on the command line. For example, `java -Xoptionsfile=myoptionfile.txt MyClass`

   In the options file, specify each option on a new line; you can use the '\' character as a continuation character if you want a single option to span multiple lines. Use the '#' character to define comment lines. You cannot specify **-classpath** in an options file. Here is an example of an options file:

   ```
   #My options file
   -X<option1>
   -X<option2>=\
   <value1>,\
   <value2>
   -D<sysprop1>=<value1>
   ```
3. **IBM_JAVA_OPTIONS** environment variable. You can set command-line options using this environment variable. The options that you specify with this environment variable are added to the command line when a JVM starts in that environment.

   For example, `set IBM_JAVA_OPTIONS=-X<option1> -X<option2>=<value1>`

General command-line options

Use these options to print help on assert-related options, set the search path for application classes and resources, print a usage method, identify memory leaks inside the JVM, print the product version and continue, enable verbose output, and print the product version.

-assert
Prints help on assert-related options.

-cp, -classpath <directories and compressed or jar files separated by : (; on Windows)>
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 (.).

-help, -?
Prints a usage message.

-memorycheck[:<*option*>**]**
Identifies memory leaks inside the JVM using strict checks that cause the JVM to exit on failure. If no option is specified, **all** is used by default. Options are:

- **all**
  - The default if just **-memorycheck** is used. Enables checking of all allocated and freed blocks on every free and allocate call. This check of the heap is the most thorough and should cause the JVM to exit on nearly all memory-related problems very soon after they are caused. This option has the greatest affect on performance.
- **callsite=**<*number of allocations*>
  - Prints callsite information every <*number of allocations*>. Deallocations are not counted. Callsite information is presented in a table with separate information for each callsite. Statistics include the number and size of allocation and free requests since the last report, and the number of the allocation request responsible for the largest allocation from each site. Callsites are presented as sourcefile:linenumber for C code and assembly function name for assembler code.

    Callsites that do not provide callsite information are accumulated into an "unknown" entry.
- **failat=**<*number of allocations*>
  - Causes memory allocation to fail (return NULL) after <*number of allocations*>. Setting <*number of allocations*> to 13 will cause the 14th allocation to return NULL. Deallocations are not counted. Use this option to ensure that JVM code reliably handles allocation failures. This option is useful for checking allocation site behavior rather than setting a specific allocation limit.
- **nofree**
  - Keeps a list of already used blocks instead of freeing memory. This list is checked, along with currently allocated blocks, for memory corruption on every allocation and deallocation. Use this option to detect a dangling pointer (a pointer that is "dereferenced" after its target memory is freed). This option cannot be reliably used with long-running applications (such as WAS), because "freed" memory is never reused or released by the JVM.
- **quick**

– Enables block padding only. Used to detect basic heap corruption. Pads every allocated block with sentinel bytes, which are verified on every allocate and free. Block padding is faster than the default of checking every block, but is not as effective.

- **skipto=**<*number of allocations*>
  – Causes the program to check only on allocations that occur after <*number of allocations*>. Deallocations are not counted. Used to speed up JVM startup when early allocations are not causing the memory problem. As a rough estimate, the JVM performs 250+ allocations during startup.
- **zero**
  – Newly allocated blocks are set 0 instead of being filled with the `0xE7E7xxxxxxxxE7E7` pattern. Setting to 0 helps you to determine whether a callsite is expecting zeroed memory (in which case the allocation request should be followed by `memset(pointer, 0, size)`).

-showversion
Prints product version and continues.

-verbose:<*option*>**[,**<*option*>**...]**
Enables verbose output. Separate multiple options using commas. These options are available:

class
Writes an entry to stderr for each class that is loaded.

dynload
Provides detailed information as each bootstrap class is loaded by the JVM:

- 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 on a Windows platform follows:

```
<Loaded java/lang/String from C:\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>
```

gc Provide verbose garbage collection information.

init
Writes information to stderr describing JVM initialisation and termination.

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 product version.

System property command-line options

Use the system property command-line options to set up your system.

-D<*name*>=<*value*>
Sets a system property.

-DCLONE_HASHTABLE_FOR_SYNCHRONIZATION
Deadlocks can occur when serializing multiple java.util.Hashtables that refer to each other in different threads at the same time. Using this command line option can resolve the deadlock, by forcing a copy of every java.util.Hashtable before this hashtable is serialized. Because this process requires temporary storage, and consumes additional processing power, the option is not enabled by default.

-Dcom.ibm.lang.management.verbose
Enables verbose information from java.lang.management operations to be written to the console during VM operation.

-Dcom.ibm.nio.DirectByteBuffer.AggressiveMemoryManagement=true
Use this property to increase dynamically the native memory limit for Direct Byte Buffers, based on their usage. This option is applicable when a Java application uses many Direct Byte Buffer objects, but cannot predict the maximum native memory consumption of the objects. Do not use the **-Xsun.nio.MaxDirectMemorySize** option with this property.

-Dcom.ibm.tools.attach.enable=yes
Enable the Attach API for this application. The Attach API allows your application to connect to a virtual machine. Your application can then load an agent application into the virtual machine. The agent can be used to perform tasks such as monitoring the virtual machine status.

-Dibm.jvm.bootclasspath
The value of this property is used as an additional search path, which is inserted between any value that is defined by **-Xbootclasspath/p:** and the bootclass path. The bootclass path is either the default or the one that you defined by using the **-Xbootclasspath:** option.

-Dibm.stream.nio=[true | false]
From v1.4.1 onwards, by default the IO converters are used. This option addresses the ordering of IO and NIO converters. When this option is set to true, the NIO converters are used instead of the IO converters.

-Djava.compiler=[NONE | j9jit23]
Disables the Java compiler by setting to NONE. Enable JIT compilation by setting to j9jit23 (Equivalent to **–Xjit**).

-Djava.net.connectiontimeout=[n]
'n' is the number of seconds to wait for the connection to be established with the server. If this option is not specified in the command line, the default value of 0 (infinity) is used. The value can be used as a timeout limit when an asynchronous java-net application is trying to establish a connection with its server. If this value is not set, a java-net application waits until the default connection timeout value is met. For instance, `java -Djava.net.connectiontimeout=2 TestConnect` causes the java-net client application to wait only 2 seconds to establish a connection with its server.

-Dsun.net.client.defaultConnectTimeout=<*value in milliseconds*>
Specifies the default value for the connect timeout for the protocol handlers used by the java.net.URLConnection class. The default value set by the protocol handlers is -1, which means that no timeout is set.

When a connection is made by an applet to a server and the server does not respond properly, the applet might seem to hang and might also cause the browser to hang. This apparent hang occurs because there is no network

connection timeout. To avoid this problem, the Java Plug-in has added a default value to the network timeout of 2 minutes for all HTTP connections. You can override the default by setting this property.

-Dsun.net.client.defaultReadTimeout=<*value in milliseconds*>
Specifies the default value for the read timeout for the protocol handlers used by the java.net.URLConnection class when reading from an input stream when a connection is established to a resource. The default value set by the protocol handlers is -1, which means that no timeout is set.

-Dsun.nio.MaxDirectMemorySize=<*value in bytes*>
Limits the native memory size for nio Direct Byte Buffer objects to the value specified.

-Dsun.rmi.transport.tcp.connectionPool=[true | any non-null value]
Enables thread pooling for the RMI ConnectionHandlers in the TCP transport layer implementation.

-Dsun.timezone.ids.oldmapping=[true | false]
From v5.0 Service Refresh 1 onwards, the Java Virtual Machine uses new time zone identifiers that change the definitions of Eastern Standard Time (EST) and Mountain Standard Time (MST). These new definitions do not take Daylight Saving Time (DST) into account. If this property is set to true, the definitions for EST and MST revert to those that were used before v5.0 Service Refresh 1 and DST is taken into account. By default, this property is set to true.

-Dswing.useSystemFontSettings=[false]
From v1.4.1 onwards, by default, Swing programs running with the Windows Look and Feel render with the system font set by the user instead of a Java-defined font. As a result, fonts for v1.4.1 differ from those in earlier releases. This option addresses compatibility problems like these for programs that depend on the old behavior. By setting this option, v1.4.1 fonts and those of earlier releases are the same for Swing programs running with the Windows Look and Feel.

JVM command-line options

Use these options to configure your JVM. The options prefixed with **-X** are nonstandard.

For options that take a <*size*> parameter, suffix the number with ″k″ or ″K″ to indicate kilobytes, ″m″ or ″M″ to indicate megabytes, or ″g″ or ″G″ to indicate gigabytes.

For options that take a <*percentage*> parameter, use a number from 0 to 1. For example, 50% is 0.5.

Options that relate to the JIT are listed under “JIT command-line options” on page 68. Options that relate to the Garbage Collector are listed under “Garbage Collector command-line options” on page 69.

-X Displays help on nonstandard options.

-Xargencoding
You can put Unicode escape sequences in the argument list. This option is set to off by default.

-Xbootclasspath:<*directories and compressed or Java archive files separated by : (; on* **Windows***)*>
Sets the search path for bootstrap classes and resources. The default is to search for bootstrap classes and resources in the internal VM directories and .jar files.

-Xbootclasspath/a:<*directories and compressed or Java archive files separated by : (; on* **Windows***)*>
Appends the specified directories, compressed files, or jar files to the end of the bootstrap class path. The default is to search for bootstrap classes and resources in the internal VM directories and .jar files.

-Xbootclasspath/p:<*directories and compressed or Java archive files separated by : (; on* **Windows***)*>
Adds a prefix of the specified directories, compressed files, or Java archive files to the front of the bootstrap class path. Do not deploy applications that use the **-Xbootclasspath:** or the **-Xbootclasspath/p:** option to override a class in the standard API. This is because such a deployment contravenes the Java 2 Runtime Environment binary code license. The default is to search for bootstrap classes and resources in the internal VM directories and .jar files.

-Xcheck:jni[**:help**][:<*option*>=<*value*>]
Performs additional checks for JNI functions. This option is equivalent to **-Xrunjnichk**. By default, no checking is performed.

-Xclassgc
Enables dynamic unloading of classes by the JVM. This is the default behavior. To disable dynamic class unloading, use the **-Xnoclassgc** option.

-Xdbg:<*options*>
Loads debugging libraries to support the remote debugging of applications. This option is equivalent to **-Xrunjdwp**. 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
This option is deprecated. Use **-Xdbg** for debugging.

-Xdiagnosticscollector[:settings=<filename>]
Enables the Diagnostics Collector. See the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html) section on “The Diagnostics Collector” for more information. The settings option allows you to specify a different Diagnostics Collector settings file to use instead of the default `dc.properties` file in the JRE.

-Xdisablejavadump
Turns off Javadump generation on errors and signals. By default, Javadump generation is enabled.

-Xdump
See the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html) section on “Using dump agents” for more information.

-Xenableexplicitgc
Signals to the VM that calls to System.gc() trigger a garbage collection. This option is enabled by default.

-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.

-Xifa:<*on* | *off* | *force*> **(z/OS only)**
z/OS R6 can run Java applications on a new type of special-purpose assist processor called the *eServer™ zSeries Application Assist Processor* (zAAP). The zSeries Application Assist Processor is also known as an IFA (Integrated Facility for Applications).

The **-Xifa** option enables Java applications to run on IFAs if they are available. Only Java code and system native methods can be on IFA processors.

-Xiss<*size*>
Sets the initial stack size for Java threads. By default, the stack size is set to 2 KB. Use the **-verbose:sizes** option to output the value that the VM is using.

-Xjarversion
Produces output information about 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.

-Xjni:<*suboptions*>
Sets JNI options. You can use the following suboption with the **-Xjni** option:

-Xjni:arrayCacheMax=[<*size in bytes*>**| unlimited]**
Sets the maximum size of the array cache. The default size is 8096 bytes.

-Xlinenumbers
Displays line numbers in stack traces for debugging. See also **-Xnolinenumbers**. By default, line numbers are on.

-Xlog
Enables message logging. To prevent message logging, use the **-Xlog:none** option. By default, logging is enabled. This option is available from Java 5 SR10. See Messages.

-Xlp<*size*> **(AIX®, Windows, and Linux® (x86, PPC32, PPC64, AMD64, EM64T))**
z/OS: Large page support requires z/OS V1.9 or later with APAR OA25485 and a System z10™ processor or later. A system programmer must configure z/OS for large pages. Users who require large pages must be authorized to the IARRSM.LRGPAGES resource in the RACF (or an equivalent security product) FACILITY class with read authority.

-Xmso<*size*>
Sets the initial stack size for operating system threads. By default, this option is set to 256 KB, except for Windows 32-bit, which is set to 32 KB.

-Xmxcl<*number*>
Sets the maximum number of class loaders. See the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html) section on "The parent-delegation model" for a description of a problem that can occur on some JVMs if this number is set too low.

-Xnoagent
Disables support for the old JDB debugger.

-Xnoclassgc
Disables dynamic class unloading. This option disables the release of native and Java heap storage associated with Java class loaders and classes that are no longer being used by the JVM. The default behavior is as defined by **-Xclassgc**. Enabling this option is not recommended except under the direction

of the IBM Java support team. This is because the option can cause unlimited native memory growth, leading to out-of-memory errors.

-Xnolinenumbers
Disables the line numbers for debugging. See also **-Xlinenumbers**. By default, line number are on.

-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, except for z/OS.

-Xoptionsfile=<*file*>

Specifies a file that contains JVM options and definitions. By default, no option file is used.

The options file does not support these options:

- **-version**
- **-showversion**
- **-fullversion**
- **-Xjarversion**
- **-memorycheck**
- **-assert**
- **-help**

<*file*> contains options that are processed as if they had been entered directly as command-line options. For example, the options file might contain:

```
-DuserString=ABC123
-Xmx256MB
```

Some options use quoted strings as parameters. Do not split quoted strings over multiple lines using the line continuation character '\'. The '¥' character is not supported as a line continuation character. For example, the following example is not valid in an options file:

```
-Xevents=vmstop,exec="cmd /c \
echo %pid has finished."
```

The following example is valid in an options file:

```
-Xevents=vmstop, \
exec="cmd /c echo %pid has finished."
```

-Xoss<*size*>
Recognized but deprecated. Use **-Xss** and **-Xmso**. Sets the maximum Java stack size for any thread. The default for AIX is 400 KB.

-Xrdbginfo:<*host*>:<*port*>
Loads the remote debug information server with the specified host and port. By default, the remote debug information server is disabled.

-Xrs
Reduces the use of operating system signals, preventing the JVM from installing signal handlers for all but exception type signals (such as SIGSEGV, SIGILL, SIGFPE). By default, the VM makes full use of operating system signals.

Note: Linux always uses SIGU3R1 and SIGU3R2.

-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.

-Xrunjdwp[:help] | [:<*option*>**=**<*value*>**, ...]**
Loads debugging libraries to support the remote debugging of applications. This option is the same as **-Xdbg**.

-Xrunjnichk[:help] | [:<*option*>**=**<*value*>**, ...]**
Deprecated. Use **-Xcheck:jni** instead.

-Xscmx<*size*>
Specifies cache size. This option applies only if a cache is being created and no cache of the same name exists. The 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 platform-dependent. The size of cache that you can specify is limited by the amount of physical memory and paging space available to the system. 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 reduces the size of the shared classes cache that you can create.

-Xshareclasses:<*suboptions*>

Enables class sharing. This option 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.

You can use the following suboptions with the **-Xshareclasses** option:

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 virtual machines 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 virtual machines using it have shut down, and the user has sufficient permissions.

expire=<*time in minutes*> **(Utility option)**
Destroys all caches that have been unused for the time specified before loading shared classes. This option is not a utility option because it does not cause the JVM to exit.

groupAccess
Sets operating system permissions on a new cache to allow group access to the cache. The default is user access only.

help
Lists all the command-line options.

listAllCaches (Utility option)
Lists all the caches on the system, describing if they are in use and when they were last used.

modified=<*modified context*>
Used when a JVMTI agent is installed that might modify bytecode at run time. If you do not specify this suboption and a bytecode modification agent is installed, classes are 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 JVMs using context *myModification1* can share the same classes. For instance, if you run an application with a modification context and then run it again with a different modification context, all classes are stored twice in the cache. See the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html) section "Dealing with runtime bytecode modification" for more information.

name=<*name*>
Connects to a cache of a given name, creating the cache if it does not exist. This option is also used to indicate the cache that is to be modified by cache utilities; for example, **destroy**. Use the **listAllCaches** utility 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 inserts the current user name. You can specify "%g" in the cache name to insert the current group name.

none
Added to the end of a command line, disables class data sharing. This suboption overrides class sharing arguments found earlier on the command line.

nonfatal
Allows the JVM to start even if class data sharing fails. Normal behavior for the JVM is to refuse to start if class data sharing fails. If you select **nonfatal** and the shared classes cache fails to initialize, the JVM starts without class data sharing.

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 with a reference to the location from which it was loaded. See the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html) section on the "printAllStats utility" for more information.

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. See the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html) section on the "printStats utility" for more information.

safemode
Forces the JVM to load all classes from disk and apply the modifications to those classes (if applicable). See the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html) section on "Using the safemode option" for more information.

silent
Disables all shared class messages, including error messages. Unrecoverable error messages, which prevent the JVM from initializing, are displayed.

verbose
Gives detailed output on the cache I/O activity, listing information about classes being stored and found. Each class loader is given a unique ID (the bootstrap loader is always 0) and the output shows the class loader hierarchy at work, where class loaders must ask their parents for a class before they can load it themselves. It is typical to see many failed requests; this behavior is expected for the class loader hierarchy. The standard option **-verbose:class** also enables class sharing verbose output if class sharing is enabled.

verboseHelper
Enables verbose output for the Java Helper API. This output shows you how the Helper API is used by your ClassLoader.

verboseIO
Gives detailed output on the cache I/O activity, listing information about classes being stored and found. Each class loader is given a unique ID (the bootstrap loader is always 0) and the output shows the class loader hierarchy at work, where class loaders must ask their parents for a class before they can load it themselves. It is typical to see many failed requests; this behavior is expected for the class loader hierarchy.

-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.

-Xss<*size*>
Sets the maximum stack size for Java threads. The default is 256 KB for 32-bit JVMs and 512 KB for 64-bit JVMs.

-Xssi<*size*>
Sets the stack size increment for Java threads. When the stack for a Java thread becomes full it is increased in size by this value until the maximum size (**-Xss**) is reached. The default is 16 KB.

-Xthr:minimizeUserCPU
Minimizes user-mode CPU usage in thread synchronization where possible. The reduction in CPU usage might be a trade-off in exchange for lower performance.

-Xtrace[:help] | [:<*option*>=<*value*>, ...]
See the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html) section on the “Controlling the trace” for more information.

-Xverify[:<*option*>**]**
With no parameters, enables the verifier, which is the default. Therefore, if used on its own with no parameters, for example, **-Xverify**, this option does nothing. Optional parameters are as follows:

- **all** - enable maximum verification
- **none** - disable the verifier

- **remote** - enables strict class-loading checks on remotely loaded classes

The verifier is on by default and must be enabled for all production servers. Running with the verifier off is not a supported configuration. If you encounter problems and the verifier was turned off using **-Xverify:none**, remove this option and try to reproduce the problem.

-XX command-line options

JVM command-line options that are specified with -XX are not stable and are not recommended for casual use.

These options are subject to change without notice.

-XXallowvmshutdown:[false | true]
This option is provided as a workaround for customer applications that cannot shutdown cleanly, as described in APAR IZ59734. Customers who need this workaround should use **-XXallowvmshutdown:false**. The default option is **-XXallowvmshutdown:true** for Java 5 SR10 onwards.

-XX:-StackTraceInThrowable
This option removes stack traces from exceptions. By default, stack traces are available in exceptions. Including a stack trace in exceptions requires walking the stack and that can affect performance. Removing stack traces from exceptions can improve performance but can also make problems harder to debug.

When this option is enabled, Throwable.getStackTrace() returns an empty array and the stack trace is displayed when an uncaught exception occurs. Thread.getStackTrace() and Thread.getAllStackTraces() are not affected by this option.

JIT command-line options

Use these JIT compiler command-line options to control code compilation.

You might need to read the section on JIT and AOT problems in the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html) to understand some of the references that are given here.

-Xcodecache<*size*>
Sets the size of each block of memory that is allocated to store native code of compiled Java methods. By default, this size is selected internally according to the CPU architecture and the capability of your system. If profiling tools show significant costs in trampolines (JVMTI identifies trampolines in a methodLoad2 event), that is a good prompt to change the size until the costs are reduced. Changing the size does not mean always increasing the size. The option provides the mechanism to tune for the right size until hot interblock calls are eliminated. A reasonable starting point to tune for the optimal size is (totalNumberByteOfCompiledMethods * 1.1). This option is used to tune performance.

-Xcomp (z/OS only)
Forces methods to be compiled by the JIT compiler on their first use. The use of this option is deprecated; use **-Xjit:count=0** instead.

-Xint
Makes the JVM use the Interpreter only, disabling the Just-In-Time (JIT) compilers. By default, the JIT compiler is enabled.

-Xjit[:<*parameter*>**=**<*value*>**, ...]**
With no parameters, enables the JIT compiler. The JIT compiler is enabled by default, so using this option on its own has no effect. Use this option to control the behavior of the JIT compiler. Useful parameters are:

count=<*n*>
Where <n> is the number of times a method is called before it is compiled. For example, setting `count=0` forces the JIT compiler to compile everything on first execution.

limitFile=(<*filename*>**,** <*m*>**,** <*n*>**)**
Compile only the methods listed on lines <*m*> to <*n*> in the specified limit file. Methods not listed in the limit file and methods listed on lines outside the range are not compiled.

optlevel=[noOpt | cold | warm | hot | veryHot | scorching]
Forces the JIT compiler to compile all methods at a specific optimization level. Specifying **optlevel** might have an unexpected effect on performance, including lower overall performance.

verbose
Displays information about the JIT and AOT compiler configuration and method compilation.

-Xquickstart
Causes the JIT compiler to run with a subset of optimizations. This quicker compilation allows for improved startup time. **-Xquickstart** can degrade performance if it is used with long-running applications that contain hot methods. The implementation of **-Xquickstart** is subject to change in future releases. By default, quickstart is disabled and JIT compilation is not delayed.

–XsamplingExpirationTime<*time*> **(from Service Refresh 5)**
Disables the JIT sampling thread after <*time*> seconds. When the JIT sampling thread is disabled, no CPU cycles are consumed by an idle JVM.

Garbage Collector command-line options

Use these Garbage Collector command-line options to control garbage collection.

You might need to read the section on “Memory management” in the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html) to understand some of the references that are given here.

The **-verbose:gc** option detailed in the section on “-verbose:gc logging” in the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html) is the main diagnostic aid that is available for runtime analysis of the Garbage Collector. However, additional command-line options are available that affect the behavior of the Garbage Collector and might aid diagnostics.

For options that take a <*size*> parameter, suffix the number with ″k″ or ″K″ to indicate kilobytes, ″m″ or ″M″ to indicate megabytes, or ″g″ or ″G″ to indicate gigabytes.

For options that take a <*percentage*> parameter, you should use a number from 0 to 1, for example, 50% is 0.5.

-Xalwaysclassgc
Always perform dynamic class unloading checks during global collection. The default behavior is as defined by **-Xclassgc**.

-Xclassgc
Enables the collection of class objects only on class loader changes. This behavior is the default.

-Xcompactexplicitgc
Enables full compaction each time System.gc() is called.

-Xcompactgc
Compacts on all garbage collections (system and global).

The default (no compaction option specified) makes the GC compact based on a series of triggers that attempt to compact only when it is beneficial to the future performance of the JVM.

-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>
This option determines the usage of which area, LOA (Large Object Area) or SOA (Small Object Area), is metered and hence which allocations are taxed during concurrent mark. Using **-Xconmeter:soa** (the default) applies the allocation tax to allocations from the small object area (SOA). Using **-Xconmeter:loa** applies the allocation tax to allocations from the large object area (LOA). If **-Xconmeter:dynamic** is specified , the collector dynamically determines which area to meter based on which area is exhausted first, whether it is the SOA or the LOA.

-Xdisableexcessivegc
Disables the throwing of an OutOfMemory exception if excessive time is spent in the GC.

-Xdisableexplicitgc

Disables System.gc() calls.

Many applications still make an excessive number of explicit calls to System.gc() to request garbage collection. In many cases, these calls degrade performance through premature garbage collection and compactions. However, you cannot always remove the calls from the application.

The **-Xdisableexplicitgc** parameter allows the JVM to ignore these garbage collection suggestions. Typically, system administrators use this parameter in applications that show some benefit from its use.

By default, calls to System.gc() trigger a garbage collection.

-Xdisablestringconstantgc
Prevents strings in the string intern table from being collected.

-Xenableexcessivegc
If excessive time is spent in the GC, this option returns null for an allocate request and thus causes an OutOfMemory exception 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. This behavior is the default.

-Xgc:<*options*>
Passes options such as verbose, compact, and nocompact to the Garbage Collector.

-Xgcpolicy:<optthruput | optavgpause | gencon | subpool (AIX, Linux PPC, zSeries, i5/OS®, and z/OS) >
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. Disables concurrent mark.

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. Enables concurrent mark.

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 subpool option (AIX, Linux PPC and zSeries, z/OS and i5/OS) uses an improved object allocation algorithm to achieve better performance when allocating objects on the heap. This option might improve performance on large SMP systems.

-Xgcthreads<*number*>
Sets the number of threads that the Garbage Collector uses for parallel operations. This total number of GC threads is composed of one application thread with the remainder being dedicated GC threads. By default, the number is set to the number of physical CPUs present. To set it to a different number (for example 4), use **-Xgcthreads4**. The minimum valid value is 1, which disables parallel operations, at the cost of performance. No advantage is gained if you increase the number of threads above the default setting; you are recommended not to do so.

On systems running multiple JVMs or in LPAR environments where multiple JVMs can share the same physical CPUs, you might want to restrict the number of GC threads used by each JVM so that, if multiple JVMs perform garbage collection at the same time, the total number of parallel operation GC threads for all JVMs does not exceed the number of physical CPUs present.

-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.

-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.

-Xloainitial<*percentage*>
Specifies the initial percentage (between 0 and 0.95) of the current tenure space allocated to the large object area (LOA). The default is 0.05 or 5%.

-Xloamaximum<*percentage*>
Specifies the maximum percentage (between 0 and 0.95) of the current tenure space allocated to the large object area (LOA). The default is 0.5 or 50%.

-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 by 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 10MB. By default, there is no maximum expansion size.

-Xmaxf<*percentage*>
Specifies the maximum percentage of heap that must be free after a garbage collection. If the free space exceeds this amount, the JVM tries to shrink the heap. The default value is 0.6 (60%).

-Xmaxt<*percentage*>
Specifies the maximum percentage of time to be spent in Garbage Collection. If the percentage of time rises above this value, the JVM tries to expand the heap. The default value is 13%.

-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. Use the **-verbose:sizes** option to produce the value that the VM is using.

-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. Use the **-verbose:sizes** option to produce the value that the VM is using.

-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<*percentage*>
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. The default value is 30%.

-Xmint<*percentage*>
Specifies the minimum percentage of time which should be spent in Garbage Collection. If the percentage of time drops below this value, the JVM tries to shrink the heap. The default value is 5%.

-Xmn<*size*>
Sets the initial and maximum size of the new area 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 try to set **-Xmn** with either **-Xmns** or **-Xmnx**, the VM does not start, returning an error. By default, **-Xmn** is not set. If the scavenger is disabled, this option is ignored.

-Xmns<*size*>
Sets the initial size of the new area to the specified value when using **-Xgcpolicy:gencon**. By default, this option is set to 25% of the value of the **-Xms** option or 64MB, whichever is less. This option returns an error if you try to use it with **-Xmn**. You can use the **-verbose:sizes** option to find out the values that the VM is currently using. If the scavenger is disabled, this option is ignored.

-Xmnx<*size*>
Sets the maximum size of the new area to the specified value when using **-Xgcpolicy:gencon**. By default, this option is set to 25% of the value of the **-Xmx** option or 64MB, whichever is less. This option returns an error if you try to use it with **-Xmn**. You can use the **-verbose:sizes** option to find out the values that the VM is currently using. If the scavenger is disabled, this option is ignored.

-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 try to set **-Xmo** with either **-Xmos** or **-Xmox**, the VM does not start, returning an error. By default, **-Xmo** is not set.

-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, set by **-Xmine** and **-Xminf**.

-Xmos<*size*>
Sets the initial size of the old (tenure) heap to the specified value when using **-Xgcpolicy:gencon**. By default, this option is set to the value of the **-Xms** option minus the value of the **-Xmns** option. This option returns an error if you try to use it with **-Xmo**. You can use the **-verbose:sizes** option to find out the values that the VM is currently using.

-Xmox<*size*>
Sets the maximum size of the old (tenure) heap to the specified value when using **-Xgcpolicy:gencon**. By default, this option is set to the same value as the **-Xmx** option. This option returns an error if you try to use it with **-Xmo**. You can use the **-verbose:sizes** option to find out the values that the VM is currently using.

-Xmr<*size*>
Sets the size of the Garbage Collection ″remembered set″. This set is a list of objects in the old (tenured) heap that have references to objects in the new area. By default, this option is set to 16K.

-Xmrx<*size*>
Sets the remembered maximum size setting.

-Xms<*size*>
Sets the initial Java heap size. You can also use the **-Xmo** option detailed above. The minimum size is 8 KB.

If scavenger is enabled, **-Xms** >= **-Xmn** + **-Xmo**.

If scavenger is disabled, **-Xms** >= **-Xmo**.

-Xmx<*size*>
Sets the maximum memory size (**-Xmx** >= **-Xms**)

Examples of the use of **-Xms** and **-Xmx**:

-Xms2m -Xmx64m
Heap starts at 2 MB and grows to a maximum of 64 MB.

-Xms100m -Xmx100m
Heap starts at 100 MB and never grows.

-Xms20m -Xmx1024m
Heap starts at 20 MB and grows to a maximum of 1 GB.

-Xms50m
Heap starts at 50 MB and grows to the default maximum.

-Xmx256m
Heap starts at default initial value and grows to a maximum of 256 MB.

-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 as defined by **-Xclassgc**. By default, class garbage collection is performed.

-Xnocompactexplicitgc
Disables compaction on System.gc() calls. Compaction takes place on global garbage collections if you specify **-Xcompactgc** or if compaction triggers are met. By default, compaction is enabled on calls to System.gc().

-Xnocompactgc
Disables compaction on all garbage collections (system or global). By default, compaction is enabled.

-Xnoloa
Prevents allocation of a large object area; all objects will be allocated in the SOA. See also **-Xloa**.

-Xnopartialcompactgc
Disables incremental compaction. See also **-Xpartialcompactgc**.

-Xpartialcompactgc
Enables incremental compaction. See also **-Xnopartialcompactgc**. By default, this option is not set, so all compactions are full.

-Xsoftmx<*size*> **(AIX only)**
This option sets the initial maximum size of the Java heap. Use the **-Xmx** option to set the maximum heap size. Use the AIX DLPAR API in your application to alter the heap size limit between **-Xms** and **-Xmx** at run time. By default, this option is set to the same value as **-Xmx**.

-Xsoftrefthreshold<*number*>
Sets the number of GCs after which a soft reference is cleared if its referent has not been marked. The default is 32, meaning that on the 32nd GC where the referent is not marked the soft reference is cleared.

-Xtgc:<*arguments*>
Provides GC tracing options, where <*arguments*> is a comma-separated list containing one or more of the following arguments:

backtrace
Before a garbage collection, a single line is printed containing the name of the master thread for garbage collection, as well as the value of the osThread slot in its J9VMThread structure.

compaction
Prints extra information showing the relative time spent by threads in the “move” and “fixup” phases of compaction

concurrent
Prints extra information showing the activity of the concurrent mark background thread

dump
Prints a line of output for every free chunk of memory in the system, including ″dark matter″ (free chunks that are not on the free list for some

reason, typically because they are too small). Each line contains the base address and the size in bytes of the chunk. If the chunk is followed in the heap by an object, the size and class name of the object is also printed. Similar to **terse**.

freeList
Before a garbage collection, prints information about the free list and allocation statistics since the last GC. Prints the number of items on the free list, including ″deferred″ entries (with the scavenger, the unused space is a deferred free list entry). For TLH and non-TLH allocations, prints the total number of allocations, the average allocation size, and the total number of bytes discarded during allocation. For non-TLH allocations, also included is the average number of entries that were searched before a sufficiently large entry was found.

parallel
Produces statistics on the activity of the parallel threads during the mark and sweep phases of a global GC.

references
Prints extra information every time that a reference object is enqueued for finalization, showing the reference type, reference address, and referent address.

scavenger
Prints extra information after each scavenger collection. A histogram is produced showing the number of instances of each class, and their relative ages, present in the survivor space. The space is linearly walked to achieve this.

terse
Dumps the contents of the entire heap before and after a garbage collection. For each object or free chunk in the heap, a line of trace output is produced. Each line contains the base address, ″a″ if it is an allocated object, and ″f″ if it is a free chunk, the size of the chunk in bytes, and, if it is an object, its class name.

-Xverbosegclog[:<*file*>**[,<X>,<Y>]]**

Causes **-verbose:gc** output to be written to the specified file. If the file cannot be found, **-verbose:gc** tries to create the file, and then continues as normal if it is successful. If it cannot create the file (for example, if an invalid filename is passed into the command), it redirects the output to stderr.

If you specify <*X*> and <*Y*> the **-verbose:gc** output is redirected to X files, each containing Y GC cycles.

The dump agent tokens can be used in the filename. See the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html) section on the “Dump agent tokens” for more information. If you do not specify <*file*>, `verbosegc.%Y%m%d.%H%M%S.%pid.txt` is used.

By default, no verbose GC logging occurs.

Appendix B. Known limitations

Known limitations on the SDK and Runtime Environment for z/OS.

If you find a problem, see the "Hints and Tips" pages, at http://www.ibm.com/servers/eserver/zseries/software/java/javafaq.html.

If you find a problem that you have been unable to solve after looking through the "Hints and Tips" pages, see http://www.ibm.com/servers/eserver/zseries/software/java/services.html for advice and information about how to raise problems.

You can find more help with problem diagnosis in the *Diagnostics Guide* at http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html.

Limitation on class path length

If there are more than 2031 characters in your class path, the shell truncates your class path to 2031 characters. If you need a class path longer than 2031 characters, use the extension class loader option to refer to directories containing your `.jar` files. For example:

`-Djava.ext.dirs=<directory>`

Where <*directory*> is the directory containing your `.jar` files.

XSLT namespace and Netbeans 5.0 problems

If the input to your transformation is a DOM that you have created programmatically, the XSLT interpreter processor might have problems with implicit namespaces. The problems are incorrect namespace declarations, or the omission of namespace declarations from the resulting document. An example Java fragment follows:

```
// Example of an explicit namespace - an attribute node will be created in the DOM for xmlns='ht
 String data = "<projectxmins='http://my.org/project/>";
 Document doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(new InputSource(r

// Example of an implicit namespace - no attribute node is created for the implicit namespace xm
 Element typeElem = doc.createElementNS("http://your.org/project", "type");
 doc.getDocumentElement().appendChild(typeElem);
```

To work around this limitation you can use the XSLT compiler processor, XSLTC. You can specify the compiler processor by using the **-XSLTC** option with the **Process** command or by setting the **javax.xml.transform.TransformerFactory** service provider to **org.apache.xalan.xsltc.trax.TransformerFactoryImpl.**

Netbeans 5.0 does not run under the JVM with default settings. To enable Netbeans to run, set the **javax.xml.transform.TransformerFactory=org.apache.xalan.xsltc.trax.TransformerFactoryImpl** property in `jre/lib/jaxp.properties`.

JConsole monitoring tool Local tab

In the IBM 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 start the application that you want to monitor, set these command-line options:

-Dcom.sun.management.jmxremote.port=<*value*>
Specifies the port the management agent listens on.

-Dcom.sun.management.jmxremote.authenticate=false
Disables authentication unless you have created a user name file.

-Dcom.sun.management.jmxremote.ssl=false
Disables SSL encryption.

JConsole monitoring tool when the JIT is not enabled

The IBM JConsole tool has the following limitations when the JIT is not enabled:

- The **Summary** tab is not available.
- The **VM** tab is not available.
- The chart accessed from the **Memory** tab is not updated.
- The chart accessed from the **Threads** tab is not updated.
- You cannot determine if the **Perform GC** button accessed from the **Memory** tab has an effect.

Incorrect stack traces when loading new classes after an Exception is caught

If new classes are loaded after an Exception has been caught, the stack trace contained in the Exception might become incorrect. The stack trace becomes incorrect if classes in the stack trace are unloaded, and new classes are loaded into their memory segments.

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().

NullPointerException with the GTK Look and Feel

DBCS environments only

If your application fails with a NullPointerException using the GTK Look and Feel, unset the **GNOME_DESKTOP_SESSION_ID** environment variable.

ASCII to EBCDIC

Because z/OS uses the EBCDIC character encoding instead of the more common ASCII encoding, sometimes there are portability problems with Java code written on z/OS. Inside the scope of the JVM, all character and string data is stored and manipulated in Unicode. I/O data outside of the virtual machine, such as on a disk or on a network, is converted to the native platform encoding. However, Java applications that implicitly assume ASCII in specific situations might require some alterations to run as expected under z/OS. For example, a platform-neutral application might have hard-coded dependencies, such as literals in ASCII.

The Java language contains the abstractions necessary to handle the switch between character encoding. The various Reader and Writer classes in the java.io package provide alternate constructors with a specified code page. This mechanism is used for globalization support, and it can also be used to force ASCII (or other) I/O where required. Not all I/O needs to be overridden; for example, character output to the display remains in the native encoding.

In addition to the Reader and Writer classes, there are a few specific situations that might require additional care. For example, the String class has an overloaded getBytes() method that takes an encoding as an additional parameter. This is useful for direct string manipulation when you are implementing custom data streams or network protocols directly in Java.

In general, straightforward workarounds are available for character encoding problems. Some encoding problems are not visible to the application because they are handled in programs running on z/OS. An example is when using Java Database Connectivity (JDBC).

IPv6 multicast support

z/OS V1R6 currently does not support IPv4-mapped Multicast addresses. If you are using an IPv4 Multicast address, you cannot join a Multicast group unless you disable IPv6 support by setting the **java.net.preferIPv4Stack** property to true.

Use the following instruction to set the property on the command line:

```
java -Djava.net.preferIPv4Stack=true <classname>
```

Using -Xshareclasses:destroy during JVM startup

When running the command java **-Xshareclasses:destroy** on a shared cache that is being used by a second JVM during startup, you might have the following issues:

- The second JVM fails.
- The shared cache is deleted.

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:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 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.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

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

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.

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

Printed in USA