



**IBM 64-bit SDK for z/OS, Java 2
Technology Edition, Version 1.4
SDK and Runtime Environment User
Guide**

Copyright information

Note: Before using this information and the product it supports, be sure to read the general information under Notices.

This edition of the User Guide applies to the IBM 64-bit SDK for z/OS, Java 2 Technology Edition, Version 1.4, and to all subsequent releases, modifications, and service refreshes, until otherwise indicated in new editions.

© Copyright Sun Microsystems, Inc. 1997, 2003, 901 San Antonio Rd., Palo Alto, CA 94303 USA. All rights reserved.

© Copyright International Business Machines Corporation, 1999, 2008. All rights reserved.

U.S. Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Preface

This User Guide describes the IBM® 64-bit SDK (Software Developer Kit) for z/OS®.

Read this User Guide if you want to use the SDK to write Sun Microsystems Java™ applications. The User Guide provides general information about the SDK and specific information about any differences in the IBM implementation of the SDK. Read this User Guide in conjunction with the more extensive documentation on the Sun Web site: <http://java.sun.com>.

The IBM JVM Diagnostics Guide provides more detailed information about the IBM JVM.

Contents

Copyright information	iii	Support for Interoperable Naming Service	22
Preface	v	System properties for tracing the ORB	23
Chapter 1. Overview	1	System properties for tuning the ORB	23
Version compatibility	1	Java 2 security permissions for the ORB	24
Upgrading the SDK	1	ORB implementation classes	24
Contents of the SDK	1	RMI over IIOP	24
Runtime Environment	2	Implementing the Connection Handler Pool for RMI	25
SDK tools	3	Chapter 7. How the JVM processes signals.	27
Tools not included in the IBM SDK	4	Signals used by the JVM	27
Chapter 2. The Java Native Interface and the Native Method Interface	5	Linking a native code driver to the signal-chaining library	28
Native formatting of Java types long, double, float	5	Chapter 8. Support for new locales	31
Chapter 3. Installing the SDK	7	Chapter 9. Enhanced BigDecimal	33
SMP and non-SMP	7	Chapter 10. Working in a multiple network stack environment.	35
Chapter 4. Configuring the SDK	9	Chapter 11. Enhanced BiDirectional support	37
Chapter 5. Launching a Java application	11	Chapter 12. Euro symbol support	39
Summary of commands	11	Chapter 13. Transforming XML documents	41
Options	11	Using an older version of Xerces or Xalan	42
Standard options	12	Chapter 14. Accessibility	43
Nonstandard options	12	Large Swing menu accessibility	43
Globalization of the java command	16	Keyboard traversal of JComboBox components in Swing	43
Working with classpaths	16	Chapter 15. Hints and tips	45
Specifying garbage collection policy	16	ASCII to EBCDIC	45
Pause time	16	Support for IPv6	45
Pause time reduction	17	Chapter 16. Reporting problems.	47
Environments with very full heaps	17	Chapter 17. Notices	49
Chapter 6. Using the SDK	19	Trademarks	50
Obtaining the IBM build and version number	19		
The Just-In-Time (JIT) compiler	19		
PATH considerations	19		
CLASSPATH considerations	20		
Debugging Java applications	20		
Java Debugger (JDB)	20		
Writing JNI applications	21		
CORBA support	22		
Support for GIOP 1.2	22		
Support for Portable Interceptors	22		

Chapter 1. Overview

The SDK for z/OS is a development environment for writing applications that conform to Sun's Java 1.4.2 Core Application Programming Interface (API).

Version compatibility

In general, any application that runs in Version 1.1.8 or 1.3.1 of the SDK should run correctly in this version.

There is no guarantee that v1.4.2-compiled classes work on pre-v1.4.0 SDK releases.

To read Sun's documentation on compatibility, see the Sun Web site at <http://java.sun.com>.

Upgrading the SDK

If you are upgrading the SDK from a previous release, back up all the configuration files and security policy files before you go ahead with the upgrade.

After the upgrade, you might have to restore or reconfigure these files because they might have been overwritten during the upgrade process. Check the syntax of the new files before restoring the original files because the format or options for the files might have changed.

The IBM 64-bit SDK for z/OS, Java 2 Technology Edition, Version 1.4 contains new versions of the IBM Java Virtual Machine and the Just-In-Time (JIT) compiler. If you are migrating from an IBM Runtime Environment on another platform, you might need to be aware of the following differences:

- The JVM shared library `libjvm.so` is installed in directory `bin/j9vm`. This is different, for example, from the IBM 31-bit SDK for z/OS, Java 2 Technology Edition, Version 1.4, where it is installed in `bin/classic`. Native applications using the JNI invocation interface might need to add the new directory path to their environment variable or variables in order to locate the JVM shared library.

Because the 64-bit z/OS SDK installation is designed to install to a different directory tree from any currently installed 31-bit SDK, both 31-bit and 64-bit SDK installations can coexist on the same system.

Contents of the SDK

The SDK contains several development tools and a Java Runtime Environment (JRE). This section describes the contents of the SDK tools and the Runtime Environment.

Applications written entirely in Java should have **no** dependencies on the IBM SDK's directory structure (or files in those directories). Any dependency on the SDK's directory structure (or the files in those directories) could result in application portability problems.

Runtime Environment

- Core Classes — These are the compiled class files for the platform and must remain zipped for the compiler and interpreter to access them. Do not modify these classes; instead, create subclasses and override where you need to.
- JRE tools — The following tools are part of the Runtime Environment and are in the `/usr/lpp/java/J1.4_64/bins/jre/bin` directory. (Where `/usr/lpp/java/J1.4_64/` is the directory in which you installed the SDK.)
 - Java Interpreter (`java`)
Runs Java classes. The Java Interpreter runs programs that are written in the Java programming language.
 - Java Interpreter (`javaw`)
Runs Java classes in the same way as the `java` command does, but does not use a console window.
 - Key and Certificate Management Tool (`keytool`)
Manages a keystore (database) of private keys and their associated X.509 certificate chains that authenticate the corresponding public keys.
 - Hardware Key and Certificate Management Tool (`hwkeytool`)
Works like `keytool` but also allows you to generate key pairs and store them in a keystore file of type JCA4758KS.
 - Policy File Creation and Management Tool (`policytool`)
Creates and modifies the external policy configuration files that define your installation's Java security policy.
 - RMI activation system daemon (`rmid`)
Starts the activation system daemon so that objects can be registered and activated in a Java virtual machine (JVM).
 - Common Object Request Broker Architecture (CORBA) Naming Service (`tnameserv`)
Starts the CORBA transient naming service.
 - Java Remote Object Registry (`rmiregistry`)
Creates and starts a remote object registry on the specified port of the current host.
 - Dump extractor (`jextract`)
Converts a system-produced dump into a common format that can be used by `jdmpview`. For more information see the relevant chapter in the *IBM JVM Diagnostics Guide* that is located at: <http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html>

Earlier versions of the IBM JRE shipped with a file called `rt.jar` in the `jre/lib` directory. From Java v1.4 onwards, this file has been replaced by multiple JAR files that reside in the `jre/lib` directory. Examples of these JAR files are:

- `core.jar`, which contains the majority of the class libraries, including the system, IO, and net class libraries
- `graphics.jar`, which contains the awt and swing class libraries
- `security.jar`, which contains the security framework code. For maintenance reasons, `security.jar` has been split up into smaller JAR files for this release.
- `server.jar`, which contains the RMI class libraries
- `xml.jar`, which contains the xml and html class libraries

This change should be completely transparent to the application. If an error is received about a missing `rt.jar` file in `CLASSPATH`, this error points to a setting that was used in Java v1.1.8 and was made obsolete in subsequent versions of Java. You can safely remove references to `rt.jar` in `CLASSPATH`.

SDK tools

- The following tools are part of the SDK and are located in the /usr/lpp/java/J1.4_64/bin directory:
 - Java Compiler (javac)
Compiles programs that are written in the Java programming language into bytecodes (compiled Java code).
 - Java Applet Viewer (appletviewer)
Tests and runs applets outside a Web browser.
 - Class File Disassembler (javap)
Disassembles compiled files and can print a representation of the bytecodes.
 - Java Documentation Generator (javadoc)
Generates HTML pages of API documentation from Java source files.
 - C Header and Stub File Generator (javah)
Enables you to associate native methods with code written in the Java programming language.
 - Java Archive Tool (jar)
Combines multiple files into a single Java Archive (JAR) file.
 - JAR Signing and Verification Tool (jarsigner)
Generates signatures for JAR files, and verifies the signatures of signed JAR files.
 - Native-To-ASCII Converter (native2ascii)
Converts a native encoding file to an ASCII file that contains characters encoded in either Latin-1 or Unicode, or both.
 - Java Remote Method Invocation (RMI) Stub Converter (rmic)
Generates stubs, skeletons, and ties for remote objects. Includes RMI over Internet Inter-ORB Protocol (RMI-IIOP) support.
 - IDL to Java Compiler (idlj)
Generates Java bindings from a given IDL file.
 - Serial Version Command (serialver)
Returns the serialVersionUID for one or more classes in a format that is suitable for copying into an evolving class.
 - Extcheck utility (extcheck)
Detects version conflicts between a target jar file and currently-installed extension jar files.
 - Cross-platform dump formatter (jdmpview)
A dump analysis tool that allows you to analyze dumps. For more information see the relevant chapter in the *IBM JVM Diagnostics Guide* that is located at: <http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html>
- INCLUDE FILES
C headers for JNI programs.
- DEMOS
The demo directory contains a number of subdirectories containing sample source code, demos, applications, and applets, that you can use.
- COPYRIGHT
Copyright notice for the SDK for z/OS software.

Note: The User Guides and the accompanying copyright files, and demo directory are the only documentation that is 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>.

Tools not included in the IBM SDK

The following tools are not included in the IBM SDK:

- MIF doclet
- orbd
- servertool

Chapter 2. The Java Native Interface and the Native Method Interface

The Java Native Interface (JNI) is included with the Runtime Environment. For information about using the JNI, see the Sun Web site: <http://java.sun.com>.

IBM does not support the Sun JDK 1.0 Native Method Interface in this release. Do not use this interface in your applications.

Native formatting of Java types long, double, float

Previous versions of the SDK for z/OS 31-bit had a set of native conversion functions and macros for formatting large Java data types. These functions and macros were:

- Function **ll2str()** - converts a jlong to an ASCII string representation of the 64-bit value.
- Function **flt2dbl()** - converts a jfloat to a jdouble.
- Macro **dbl2nat()** - converts a jdouble to an ESA/390 native double.
- Macro **dbl_sqrt()** - takes the square root of a jdouble and returns a jdouble.
- Functions **dbl2str()** and **flt2str()** - convert their arguments into an ASCII string representation.

These functions and macros are not supported by the 64-bit SDK, and are no longer part of the libjvm.x interface. However, to provide a migration path, the functions have been moved to the demos area of the SDK and the appropriate demo code for these functions has been updated to reflect the changes.

The functions **ll2str()**, **dbl2str()**, and **flt2str()** are provided in the following object files:

- `/usr/lpp/java/J1.4_64/demo/jni/JNINativeTypes/c/convert.o` (31-bit version).
- `/usr/lpp/java/J1.4_64/demo/jni/JNINativeTypes/c/convert64.o` (64-bit version).

The function **flt2dbl()** and the macros **dbl2nat()** and **dbl_sqrt()** are not defined. However, the following macros give their definitions:

```
#include <math.h>
#define flt2dbl(f) ((double)f)
#define dbl2nat(a) ((a))
#define dbl_sqrt(a) (sqrt(a))
```

Note: These functions and macros are obsolete because the latest C/C++ compilers and runtimes can convert jlong, jdouble, and jfloat data types to strings by using **printf()**-type functions.

Chapter 3. Installing the SDK

SMP and non-SMP

The following Web site has all the necessary instructions for ordering, downloading, installing, and verifying the install: <http://www-1.ibm.com/servers/eserver/zseries/software/java/>

Chapter 4. Configuring the SDK

After you install the SDK , edit your shell login script and add this directory to your **PATH** statement:

```
/usr/lpp/java/J1.4_64/bin
```

If you installed the SDK in a directory other than `/usr/lpp/java/J1.4_64/`, replace `/usr/lpp/java/J1.4_64/` with the directory in which you installed the SDK .

Chapter 5. Launching a Java application

The **java** tool launches a Java application. It requires an initial Java class name as parameter. If you do not supply this, a usage message is printed.

It does this by starting a Java Runtime Environment, loading a specified class, and invoking that class's main method. The method declaration must have the signature:

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

The method must be declared public and static, it must not return any value, and it must accept a String array as a parameter. By default, the first non-option argument is the name of the class to be invoked. Use a fully qualified class name.

The JVM searches for the initial class, and other classes that are used, in three sets of locations: the bootstrap classpath, the installed extensions, and the user classpath. Arguments after the class name or JAR file name are passed to the main function.

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

Summary of commands

The **java** and **javaw** command have the following syntax:

```
java [ options ] class [ arguments ... ]  
java [ options ] -jar file.jar [ arguments ... ]  
javaw [ options ] class [ arguments ... ]  
javaw [ options ] -jar file.jar [ arguments ... ]
```

Items that are within brackets are optional.

options

Command-line options.

class

Name of the class to invoke.

file.jar

Name of the jar file to invoke. It is used only with **-jar**.

argument

Argument passed to the **main** function.

If the **-jar** option is specified, the named JAR file contains class and resource files for the application, with the startup class indicated by the Main-Class manifest header.

Options

The launcher has a set of standard options that are supported on the current runtime environment and will be supported in future releases. In addition, there is a set of nonstandard options.

Standard options

- D***<property_name>=<value>*
Sets a system property.
- assert**
Prints help on assert-related options.
- cp** or **-classpath** *<directories and zip or jar files separated by :>*
Sets the search path for application classes and resources. If **-classpath** and **-cp** are not used, and **CLASSPATH** is not set, the user classpath is, by default, the current directory (.). Also see “Working with classpaths” on page 16.
- help** or **-?**
Prints a usage message.
- showversion**
Prints product version and continues.
- verbose**:*[class |gc terse |gc |dynload |stackwalk |sizes |stack |debug |jni]*
Enables verbose output.
- version**
Prints product version.
- X**
Prints help on nonstandard options.

Nonstandard options

The **-X** options listed below are nonstandard and subject to change without notice.

- Xargencoding**
Allows you to put Unicode escape sequences in the argument list.
- Xbootclasspath:***<directories and zip or jar files separated by ;>*
Sets the search path for bootstrap classes and resources.
- Xbootclasspath/a:***<directories and zip or jar files separated by ;>*
Appends the specified directories, zip, or jar files to the end of bootstrap class path.
- Xbootclasspath/p:***<directories and zip or jar files separated by ;>*
Prepends the specified directories, zip, or jar files to the front of the bootstrap class path. Do not deploy applications that use the **-Xbootclasspath:** or **-Xbootclasspath/p:** option to override a class in *core.jar*, *graphics.jar*, *server.jar*, *security.jar*, *xml.jar*, *tools.jar*, or *charsets.jar*, because such a deployment would contravene the Java 2 Runtime Environment binary code license.
- Xcheck:jni**
Performs additional checks for JNI functions. You can also use **-Xrunjniclk**.
- Xcheck:nabounds**
Performs additional checks for JNI array operations. You can also use **-Xrunjniclk**.
- Xcomp**
Forces methods to be compiled by the JIT on their first use.
- Xcompactexplicitgc**
Compact on every call to *System.gc()*. See also **-Xnocompactexplicitgc**
- Xcompactgc**
Compact every Garbage Collector cycle. See also **-Xnocompactgc**
- Xdbg:***<options>*
Loads debugging libraries to support the remote debugging of applications. This is the same as **-Xrunjdwp**

- Xdbginfo:***<path to symbol file>*
Loads and passes options to the debug information server.
- Xdisablejavadump**
Causes system dumps to be generated instead of javadumps when errors occur.
- Xdebug**
Starts the JVM with the debugger enabled. Used with **-Xrunjdw**.
- Xdisableexplicitgc**
Changes calls to `System.gc()` into no-ops.
- 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 of the Java application launcher.
- Xgc:***<options>*
Passes options such as `verbose`, `compact`, `nocompact`, and so on, to the Garbage Collector.
- Xgcpolicy:***[:optthruput] | [:optavgpause] | [:gencon]*
Controls the behavior of the garbage collector. The `optthruput` option is the default and delivers very high throughput to applications, but at the cost of occasional pauses. The `optavgpause` option reduces the time that is spent in these garbage collection pauses and limits the effect of increasing heap size on the length of the garbage collection pause. Use `optavgpause` if your configuration has a very large heap. The `gencon` option requests the combined use of concurrent and generational GC to help minimize the time that is spent in any garbage collection pause.
- Xgcthreads***<number of threads>*
Sets the number of threads that must perform garbage collection.
- Xifa:***<on | off | force>*
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 these are available. Only Java code and system native methods can be on IFA processors. For more information, see the Java Diagnostics Guide.
- Xint**
Makes the JVM use the Interpreter only, disabling the JIT and AOT.
- Xiss***<size>*
Sets the initial Java thread stack size.
- Xjit:***<number of threads>*
Enables the JIT or passes options to it, or both. See also **-Xnojit**.
- Xlinenumbers**
Enables the line numbers for debugging. See also **-Xnolinelnumbers**.
- 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.
- Xmaxf***<size>*

Specifies the maximum percentage of heap that must be free after a garbage collection. If the free space exceeds this amount, the JVM attempts to shrink the heap. Specify the size as a decimal value in the range 0-1, for example `-Xmaxf0.5` sets the maximum free space to 50%. The default value is 0.6.

-Xmca<size>

Sets RAM class segment to increment by the specified size.

-Xmco<size>

Sets ROM class segment to increment by the specified size.

-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 by 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 50MB.

-Xminf<size>

Specifies the minimum percentage of heap that should be free after a garbage collection. If the free space falls below this amount, the JVM attempts to expand the heap. Specify the size as a decimal value in the range 0-1; for example, a value of `-Xminf0.3` requests the minimum free space to be 30% of the heap

-Xmn<size>

Sets the size of the new heap to the specified size.

-Xmo<size>

Sets the initial Java heap size. You can also use **-Xms**.

-Xmoi<size>

Increments the initial Java heap size.

-Xmr<size>

Sets the remembered set size.

-Xms<size>

Sets the initial Java heap size. You can also use **-Xmo**.

-Xmso<size>

Sets the size of the OS thread stack.

-Xmx<size>

Sets maximum Java heap size.

-Xnm

Specifies the size of the monitor pool.

-Xnoagent

Disables the old sun.tools.debug interface, so that you can use the JPDA debug support.

-Xnoaot

Disables the AOT.

-Xnoclassgc

Disables class garbage collection. This switches off garbage collection of storage associated with Java classes that are no longer being used by the JVM.

-Xnocompactgc

Disables compaction for the Garbage Collector. See also **-Xcompactgc**

-Xnocompactexplicitgc

Disables compact on a call to `System.gc()`. See also **-Xcompactexplicitgc**

-Xnojit

Disables the JIT. See also **-Xjit**.

- Xnolinelnumbers**
Disables the line numbers for debugging. See also **-Xlinenumbers**.
- Xnopartialcompactgc**
Disables incremental compaction on a call to `System.gc()`. See also **-Xpartialcompactgc**
- Xnosigcatch**
Disables JVM recovery code. See also **-Xsigcatch**.
- Xnosigchain**
Disables the chaining of signal handlers. See also **-Xsigchain**.
- Xoptionsfile=<file>**
Specifies a file that contains JVM options and defines.
- Xoss<size>**
Sets maximum Java stack size for any thread.
- Xpartialcompactgc**
Enables incremental compaction on every call to `System.gc()`. See also **-Xnopartialcompactgc**
- Xquickstart**
Improves startup time by delaying compilation.
- Xrdbginfo:<host>:<port>**
Loads and passes options to the remote debug information server.
- Xrs**
Reduces the use of operating system signals.
- Xrunlibrary_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.
 - **-Xrunjdw[:help] | [:<option>=<value>, ...]**
Loads debugging libraries to support the remote debugging of applications. This is the same as **-Xdbg**
 - **-Xrunjchck[:verbose | trace | profile | nonfatal | nowarn | noadvice | pedantic | help]**
Performs additional checks for JNI functions.
- Xsigcatch**
Enables JVM recovery code. See also **-Xnosigcatch**.
- Xsigchain**
Enables the chaining of signal handlers. See also **-Xnosigchain**.
- XsigquitToFile<path to file><filename>**
Dumps the information from the sigquit trace to the specified file.
- Xss<size>**
Sets maximum native stack size for any thread.
- Xthr:<options>**
Sets the threading options.
- Xverbosegclog:<path to file><filename>[X, Y]**
Causes verboseGC output to be written to the specified file. If the file exists, it is overwritten. Otherwise, if an existing file cannot be opened or a new file cannot be created, the output is redirected to stderr. If you specify the arguments X and Y (both are integers) the verboseGC output is redirected to X number of files, each containing Y number of gc cycles worth of verboseGC output. These files have the form *filename1*, *filename2*, and so on.
- Xverify**
Enables strict class checking for every class that is loaded.

-Xverify:none
Disables strict class checking.

Globalization of the java command

The **java** command and other java launcher commands (such as **javaw**) allow a class name to be specified as any character that is in the character set of the current locale.

You can also specify any Unicode character in the class name and arguments by using java escape sequences. To do this, you must specify **-Xargencoding**. To specify a Unicode character, use escape sequences in the form `\u####`, where # is a hexadecimal digit (0 through 9, A through F).

Alternatively, to specify that the class name and command arguments are in UTF8 encoding, use **-Xargencoding:utf8**, or in ISO8859_1 encoding use **-Xargencoding:latin**.

The **java** and **javaw** commands give translated output messages. These messages differ based on the locale in which Java is running. The detailed error descriptions and other debug information that is returned by **java** are in English.

Working with classpaths

You can specify a class name as a complete file name including a full path and the .class extension. In previous versions, you could specify only the class that was relative to the **CLASSPATH**, and the .class extension was not allowed. Use of the complete file name permits you to launch a java application from your desktop or file launcher. If you specify a .class file with path and extension, the specified path is put into the **CLASSPATH**.

Specifying garbage collection policy

The **-Xgcpolicy** JVM runtime option specifies garbage collection policy.

-Xgcpolicy takes the values **optthruput** (the default), **optavgpause**, or **gencon**. The option controls garbage collector behavior, making tradeoffs 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

-Xgcpolicy:optavgpause

-Xgcpolicy:gencon

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, **optthroughput**, 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
- Generational garbage collection

The **-Xgcpolicy:optavgpause** command-line option requests the use of concurrent garbage collection to reduce significantly the time that is spent in garbage collection pauses. Concurrent GC reduces the pause time by performing some garbage collection activities concurrently with normal program execution to minimize the disruption caused by the collection of the heap. The **-Xgcpolicy:optavgpause** option also limits the effect of increasing the heap size on the length of the garbage collection pause. The **-Xgcpolicy:optavgpause** option is most useful for configurations that have large heaps. With the reduced pause time, you might experience some reduction of throughput to your applications.

During concurrent garbage collection a significant amount of time is wasted identifying relatively long-lasting objects that cannot then be collected. If the GC concentrates on just those objects that are most likely to be recyclable, you can further reduce pause times for some applications. Generational GC achieves this by dividing the heap into two "generations", the "nursery" and the "tenure" areas. Objects are placed in one of these areas depending on their age. The nursery is the smaller of the two and contains younger objects; the tenure is larger and contains older objects. Objects are first allocated to the nursery; if they survive long enough they are promoted to the tenure area eventually.

Generational GC depends on most objects not lasting long. Generational GC reduces pause times by concentrating the effort to reclaim storage on the nursery because it has the most recyclable space. Rather than occasional but lengthy pause times to collect the entire heap, the nursery is collected more frequently and, if the nursery is small enough, pause times are comparatively short. However, generational GC has the drawback that, over time, the tenure area might become full if too many objects last too long. To minimize the pause time when this situation occurs, use a combination of concurrent GC and generational GC. The **-Xgcpolicy:gencon** option requests the combined use of concurrent and generational GC to help minimize the time that is spent in any garbage collection pause.

Environments with very full heaps

If the Java heap becomes nearly full, and very little garbage is to be reclaimed, requests for new objects might not be satisfied quickly because no space is immediately available. If the heap is operated at near-full capacity, application performance might suffer regardless of which of the above options is used; and, if requests for more heap space continue to be made, the application receives an `OutOfMemory` exception, which results in JVM termination if the exception is not caught and handled. At this point the JVM will produce a "javadump" diagnostic file. In these conditions, you are recommended either to increase the heap size by using the **-Xmx** option, or to reduce the number of application objects in use.

Chapter 6. Using the SDK

The Java tools are programs that are run from a shell prompt; they do not have a Graphical User Interface (GUI).

The following sections give information about using the SDK for z/OS.

Obtaining the IBM build and version number

To obtain the IBM build and version number, at a command prompt type:

```
java -version
```

The Just-In-Time (JIT) compiler

The Just-In-Time (JIT) compiler dynamically generates machine code for frequently used bytecode sequences in Java applications and applets while they are running.

The SDK for z/OS includes the JIT, which is enabled by default. You can disable the JIT to help isolate a problem with a Java application, an applet, or the compiler itself.

To disable the JIT, use the **-Xint** option. At the command prompt window where you run the application, type:

```
java -Xint class
```

To verify whether or not the JIT is enabled, type at a command prompt:

```
java -version
```

If the JIT is in use, a message is displayed that includes:

```
(JIT enabled)
```

If the JIT is not in use, a message is displayed that includes:

```
(JIT disabled)
```

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

PATH considerations

After installing the SDK for z/OS, you can run a tool by typing its name at a shell prompt with a filename as an argument.

You can specify the path to a tool by typing the path before the name of the tool each time. For example, if the SDK for z/OS software is installed in `/usr/lpp/java/J1.4_64/bin`, you can compile a file named `myfile.java` by typing the following at a shell prompt:

```
/usr/lpp/java/J1.4_64/bin/javac myfile.java
```

To avoid typing the full path each time:

1. Add the following directories to the **PATH** environment variable:

```
/usr/lpp/java/J1.4_64/bin
```

2. Compile the file with the **javac** tool. For example, compile the file `myfile.java` by typing the following at a shell prompt:

```
javac myfile.java
```

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

```
echo $PATH
```

CLASSPATH considerations

The **CLASSPATH** tells the SDK tools, such as **java**, **javac**, and **javadoc**, where to find the Java class libraries. If you keep the **bin** and **lib** directories under the same parent directory level, the executable files can find the classes.

You need to set the **CLASSPATH** explicitly only if one of the following applies:

- You require a different library, such as one that you develop.
- You change the location of the **bin** and **lib** directories such that they no longer have the same common parent directory.
- You plan to develop or run applications that are using different runtime environments on the same system.

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

```
echo $CLASSPATH
```

If you plan to 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 plan to run multiple applications simultaneously and use different runtime environments, be sure that each application is run in its own shell.

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

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 z/OS.

Java Debugger (JDB)

The Java Debugger (JDB) is included in the SDK for z/OS. The debugger is invoked by the **jdb** command; it "attaches" to the JVM using JPDA. To debug a Java application:

1. Start the JVM with the following options:

```
java -Xdebug -Xrunjdpw:transport=dt_socket,server=y,  
address=<port number>  
MyApplication <MyApplication args>
```

2. The JVM starts up, but suspends execution before it starts the Java application. In a separate session, you can attach the debugger to the JVM:

```
jdb -attach <port number>
```

The debugger will attach to the JVM, and you can now issue a range of commands to examine and control the Java application, for example type "run" to allow the Java application to execute.

To find out more about JDB options, type:

```
jdb -help
```

To find out more about JDB commands:

1. Type `jdb`
2. At the `jdb` prompt, type `help`

You can also use JDB to debug remote Java applications:

1. Start the JVM as before.
2. Attach the debugger to the remote machine:

```
jdb -attach <machine name or ip address>:<port number>
```

When you launch a debug session using the `dt_socket` transport, be sure that the specified ports are free to use.

The Java Virtual Machine Debugging Interface (JVMDI) is *not* supported.

For more information on JDB and JPDA and their usage see the following Web sites:

- <http://java.sun.com/products/jpda/>
- <http://java.sun.com/j2se/1.4.2/docs/guide/jpda/>

Writing JNI applications

Valid JNI version numbers that native programs can specify on the `JNI_CreateJavaVM()` API call are:

- `JNI_VERSION_1_2(0x00010002)`
- `JNI_VERSION_1_4(0x00010004)`

This version number determines only the level of the JNI native interface to use. The actual level of the JVM that is created is specified by the J2SE libraries (that is, v1.4.2). The JNI interface API *does not* affect the language specification that is implemented by the JVM, the class library APIs, or any other area of JVM behavior. For further information see <http://java.sun.com/j2se/1.4.2/docs/guide/jni>.

Use the **`com.ibm.vm.bitmode`** system property to determine if you're running with a 31- or 64-bit JVM. An application that has two JNI libraries, one built for 31-bit and the other for 64-bit, can use this system property to operate with both 31- and 64-bit JVMs. At runtime the Java code can select which library to load based on the value of **`com.ibm.vm.bitmode`**.

For more information about writing 64-bit applications, see the red paper *z/OS 64-bit C/C++ and Java Programming Environment* on developerWorks[®] at <http://www-106.ibm.com/developerworks/java/jdk/index.html>.

Note: Version 1.1 of the Java Native Interface (JNI) is not supported.

CORBA support

The Java 2 Platform, Standard Edition (J2SE) supports, at a minimum, the specifications that are defined in the Official Specifications for CORBA support in J2SE V1.4 at <http://java.sun.com/j2se/1.4.2/docs/api/org/omg/CORBA/doc-files/compliance.html>. In some cases, the IBM J2SE ORB supports more recent versions of the specifications.

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*, which you can obtain from:

<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/01-09-58>

Portable Interceptors are hooks into the ORB through which ORB services can intercept the normal flow of execution of the ORB.

Support for Interoperable Naming Service

This SDK supports the Interoperable Naming Service, as defined by the OMG in the document *ptc/00-08-07*, which you can obtain from:

<http://www.omg.org/cgi-bin/doc?ptc/00-08-07>

The default port that is used by the Transient Name Server (the `tnameserv` command), when no **ORBInitialPort** parameter is given, has changed from *900* to *2809*, which is the port number that is registered with the IANA (Internet Assigned Number Authority) for a CORBA Naming Service. Programs that depend on this default might have to be updated to work with this version.

The initial context that is returned from the Transient Name Server is now an `org.omg.CosNaming.NamingContextExt`. Existing programs that narrow the reference to a context `org.omg.CosNaming.NamingContext` still work, and do not need to be recompiled.

The ORB supports the **-ORBInitRef** and **-ORBDefaultInitRef** parameters that are defined by the Interoperable Naming Service specification, and the `ORB::string_to_object` operation now supports the `ObjectURL` string formats (`corbaloc:` and `corbaname:`) that are defined by the Interoperable Naming Service specification.

The OMG specifies a method `ORB::register_initial_reference` to register a service with the Interoperable Naming Service. However, this method is not available in the Sun Java Core API at Version 1.4.2. Programs that need to register a service in the current version must invoke this method on the IBM internal ORB implementation class. For example, to register a service "MyService":

```
((com.ibm.CORBA.iiop.ORB)orb).register_initial_reference("MyService",
serviceRef);
```

where orb is an instance of org.omg.CORBA.ORB, which is returned from ORB.init(), and serviceRef is a CORBA Object, which is connected to the ORB. This mechanism is an interim one, and is not compatible with future versions or portable to non-IBM ORBs.

System properties for tracing the ORB

A runtime debug feature provides improved serviceability. You might find it useful for problem diagnosis or it might be requested by IBM service personnel. Tracing is controlled by three system properties.

- To turn on tracing, set **com.ibm.CORBA.Debug=true**.
- To format and add to the trace GIOP messages sent and received, set **com.ibm.CORBA.CommTrace=true**. By default, ORB tracing goes to files that have names of the form orbtrc.DDMMYYYY.HHmm.SS.txt.
- To send it to a different file, set **com.ibm.CORBA.Debug.Output=<filename>**.

For example, to trace events and formatted GIOP messages, type:

```
java -Dcom.ibm.CORBA.Debug=true
-Dcom.ibm.CORBA.CommTrace=true myapp
```

Do not turn on tracing for normal operation, because it might cause performance degradation. Even if you have switched off tracing, FFDC (First Failure Data Capture) is still working, so that only serious errors are reported. If a debug output file is generated, examine it to check on the problem. For example, the server might have stopped without performing an ORB.shutdown().

The content and format of the trace output might vary from version to version.

System properties for tuning the ORB

The following properties help you to tune the ORB:

- To control GIOP 1.2 fragmentation, set the system property **com.ibm.CORBA.FragmentSize**.

For example, to set the fragment size to 4096 bytes:

```
java -Dcom.ibm.CORBA.FragmentSize=4096 myapp
```

The default fragment size is 1024 bytes. You can turn off fragmentation by setting the fragment size to 0.

- In a heterogeneous network, the response to a CORBA Request might be delayed or lost. You can set the maximum time to wait by using the system property **com.ibm.CORBA.RequestTimeout**. Also, you can use **com.ibm.CORBA.LocateRequestTimeout** to control the timeout for LocateRequests. For example, to restrict both delays to 30 seconds, type:

```
java -Dcom.ibm.CORBA.RequestTimeout=30
-Dcom.ibm.CORBA.LocateRequestTimeout=30 myapp
```

By default, the ORB waits indefinitely for a response. Do not set the timeout too low, or connections might be ended unnecessarily.

- If your program is providing a service in the network, you might have to set, to a well-known value, the number of the port from which the ORB reads incoming requests. You can control this by using the system property **com.ibm.CORBA.ListenerPort**.

For example, to make the ORB use port 1050, type:

```
java -Dcom.ibm.CORBA.ListenerPort=1050 myapp
```

If this property is set, the ORB starts listening as soon as it is initialized. Otherwise, it starts listening only when required.

Java 2 security permissions for the ORB

When running with a Java 2 SecurityManager, invocation of some methods in the CORBA API classes might cause permission checks to be made, which might result in a SecurityException. Affected methods include the following:

Table 1. Methods affected when running with Java 2 SecurityManager

Class/Interface	Method	Required permission
org.omg.CORBA.ORB	init	java.net.SocketPermission resolve
org.omg.CORBA.ORB	connect	java.net.SocketPermission listen
org.omg.CORBA.ORB	resolve_initial_references	java.net.SocketPermission connect
org.omg.CORBA.portable.ObjectImpl	_is_a	java.net.SocketPermission connect
org.omg.CORBA.portable.ObjectImpl	_non_existent	java.net.SocketPermission connect
org.omg.CORBA.portable.ObjectImpl	OutputStream _request (String, boolean)	java.net.SocketPermission connect
org.omg.CORBA.portable.ObjectImpl	_get_interface_def	java.net.SocketPermission connect
org.omg.CORBA.Request	invoke	java.net.SocketPermission connect
org.omg.CORBA.Request	send_deferred	java.net.SocketPermission connect
org.omg.CORBA.Request	send_oneway	java.net.SocketPermission connect
javax.rmi.PortableRemoteObject	narrow	java.net.SocketPermission connect

If your program uses any of these methods, ensure that it is granted the necessary permissions.

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 to do distributed Java programming. RMI over IIOP (RMI-IIOP) uses the Common Object Request Broker Architecture (CORBA) standard Internet Inter-ORB Protocol (IIOP protocol) to extend the base Java RMI to perform communication. This allows direct interaction with any other CORBA Object Request Brokers (ORBs), whether they were implemented in Java or another programming language.

The following documentation is available:

- The RMI-IIOP Programmer's Guide is an introduction to writing RMI-IIOP programs. The guide is installed when you install this documentation. You can also find the guide on the Web site above under **Supporting documentation** (on the right-hand side).
- The demo directory contains:
 - A "Hello World" example that can switch between the Java Remote Method Protocol (JRMP) and IIOP protocols (demo/rmi-iiop/hello)
 - A "Hello World" example that interacts with a standard IDL program (demo/rmi-iiop/idl)
- The *Java Language to IDL Mapping* document is a detailed technical specification of RMI-IIOP and can be found at:
<http://www.omg.org/cgi-bin/doc?ptc/00-01-06.pdf>

Implementing the Connection Handler Pool for RMI

Thread pooling for RMI Connection Handlers is *not* enabled by default.

To enable the connection pooling implemented at the RMI TCPTransport level, set the option

```
-Dsun.rmi.transport.tcp.connectionPool=true (or any non-null value)
```

This version of the Runtime Environment does not have any setting that you can use to limit the number of threads in the connection pool.

For more information, see the Sun Java site: <http://java.sun.com>.

Chapter 7. 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, after the JVM has finished processing, the application handler for this signal is called.

If the signal is for a non-Java thread, and the application that installed the JVM had previously installed its own handler for the signal, control is given to that handler. Otherwise, if the signal is requested by the JVM or Java application, the signal is ignored or the default action is taken.

For exception and error signals, the JVM either:

- Handles the condition and recovers, or
- Enters a controlled shutdown sequence where it:
 1. Outputs a Javdump, to describe the JVM state at the point of failure
 2. Calls your application's signal handler for that signal
 3. Calls any application-installed abort hook
 4. Performs the necessary cleanup to give a clean shutdown

For information about writing a launcher that specifies the above hooks, see: <http://www-106.ibm.com/developerworks/java/library/i-signalhandling/>. This item was written for Java V1.3.1, but still applies to later versions.

For interrupt signals, the JVM also enters a controlled shutdown sequence, but this time it is treated as a normal termination that:

- Calls your application's signal handler for that signal
- Runs all application shutdown hooks
- Calls any application-installed exit hook
- Performs the necessary JVM cleanup

The shutdown is identical to the shutdown initiated by a call to the Java method `System.exit()`.

Other signals that are used by the JVM are for internal control purposes and do not cause it to terminate. The only control signal of interest is SIGQUIT, which causes a Javdump to be generated.

Signals used by the JVM

Table 2 on page 28 below shows the signals that are used by the JVM. The signals are grouped in the table by type or use, as follows:

- **Exceptions:** The operating system synchronously raises an appropriate exception signal whenever a fatal condition occurs.
- **Errors:** The JVM raises a SIGABRT if it detects a condition from which it cannot recover.

- **Interrupts:** Interrupt signals are raised asynchronously, from outside a JVM process, to request shutdown.
- **Controls:** Other signals that are used by the JVM for control purposes.

Table 2. Signals used by the JVM

Signal Name	Signal type	Description	Disabled by -Xrs
SIGBUS	Exception	Incorrect access to memory (data misalignment)	No
SIGSEGV	Exception	Incorrect access to memory (write to inaccessible memory)	No
SIGILL	Exception	Illegal instruction (attempt to invoke an unknown machine instruction)	No
SIGFPE	Exception	Floating point exception (divide by zero)	No
SIGABRT	Error	Abnormal termination. The JVM raises this signal whenever it detects a JVM fault.	No
SIGINT	Interrupt	Interactive attention (CTRL-C). JVM exits normally.	Yes
SIGTERM	Interrupt	Termination request. JVM will exit normally.	Yes
SIGHUP	Interrupt	Hang up. JVM exits normally.	Yes
SIGQUIT	Control	The JVM uses this for taking Java core dumps.	No
SIGPIPE	Control	Broken pipe. Set to SIG_IGN	No

Use the **-Xrs** (reduce signal usage) option to prevent the JVM from handling most signals. For more information, see Sun's Java application launcher page at <http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/java.html>.

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 services of the JVM.

Linking a native code driver to the signal-chaining library

The Runtime Environment contains signal-chaining. Signal-chaining enables the JVM to interoperate more efficiently with native code that installs its own signal handlers.

Signal-chaining enables an application to link and load the shared library `libjsig` before the system libraries. The 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.

Chapter 8. Support for new locales

From Service Refresh 8, the following new locale is added: Serbia (SE), with these three new locale variations:

- 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 9. Enhanced BigDecimal

The SDK includes an enhanced BigDecimal class (`com.ibm.math.BigDecimal`) for Java programming. It is provided (with its supporting class `MathContext`) as an alternative to the `java.math.BigDecimal` class.

The `java.math.BigDecimal` class provides a minimal, fixed-point, decimal arithmetic capability only. The `com.ibm.math.BigDecimal` class adds:

- Floating-point arithmetic
- Support for exponential notation (both scientific and engineering)
- New operators (integer division, power, and remainder)
- Formatting
- Additional conversions
- Efficient control of the arithmetic context, using simple `MathContext` objects
- More straightforward programming, with no need for manual accounting of scale, precision, and exponents, along with additional utility methods
- Improved robustness: for example, "decapitated" conversions raise an exception rather than losing the most significant digits quietly.

The `com.ibm.math.BigDecimal` class uses significantly fewer resources for common operations than the `java.math.BigDecimal` class.

The `com.ibm.math.BigDecimal` class is compatible with `java.math.BigDecimal`, and supports all of its methods. To use the `com.ibm.math.BigDecimal` class, change the import statement at the top of your `.java` file: `import java.math.*;` to `import com.ibm.math.*;`

For more information see <http://www.alphaworks.ibm.com/tech/bigdecimal>.

Chapter 10. 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.serversocket.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.serversocket.recover** property is set to *true*, Java attempts to redrive the socket connection on the new stack if listening on all 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.

Chapter 11. Enhanced BiDirectional support

The IBM SDK includes enhanced BiDirectional support. For more information, see <http://www-106.ibm.com/developerworks/java/jdk/bidirectional/index.html>.

Chapter 12. Euro symbol support

The IBM SDK sets 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.

In V1.4.2 SR6, the default for the Slovenian locale is set to the Euro. If you install SR6 before 1 January 2007, you might want to change the currency to the Tolar.

Chapter 13. Transforming XML documents

The IBM SDK contains the XSLT4J 2.6 processor and the XML4J 4.3 parser that conform to the JAXP 1.2 specification. These tools allow you to parse and transform XML documents independently from any given XML processing implementation. By using "Factory Finders" to locate the SAXParserFactory, DocumentBuilderFactory and TransformerFactory implementations, your application can swap between different implementations without having to change any code.

The XSLT4J 2.6 processor allows you to choose between the original XSLT Interpretive processor or the new XSLT Compiling processor. The Interpretive processor is designed for tooling and debugging environments and supports the XSLT extension functions that are not supported by the XSLT Compiling processor. The XSLT Compiling processor is designed for high performance runtime environments; it generates a transformation engine, or *translet*, from an XSL stylesheet. This approach separates the interpretation of stylesheet instructions from their runtime application to XML data.

The XSLT Interpretive processor is the default processor. To select the XSLT Compiling processor you can:

- Change the entry in the `jaxp.properties` file (located in `/usr/lpp/java/J1.4_64/jre/lib`), or
- Set the system property for the `javax.xml.transform.TransformerFactory` key to `org.apache.xalan.xsltc.trax.TransformerFactoryImpl`.

To implement properties in the `jaxp.properties` file, copy `jaxp.properties.sample` to `jaxp.properties` in `/usr/lpp/java/J1.4_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.XSLTCB2BDTMMManager` class as the provider of the service `org.apache.xalan.xsltc.dom.XSLTCDTMMManager`. To determine the service provider, try each step until you find `org.apache.xalan.xsltc.dom.XSLTCDTMMManager`:

1. Check the setting of the system property `org.apache.xalan.xsltc.dom.XSLTCDTMMManager`.
2. Check the value of the property `org.apache.xalan.xsltc.dom.XSLTCDTMMManager` in the file `/usr/lpp/java/J1.4_64/jre/lib/xalan.properties`.
3. Check the contents of the file `META-INF/services/org.apache.xalan.xsltc.dom.XSLTCDTMMManager` for a class name.
4. Use the default service provider, `org.apache.xalan.xsltc.dom.XSLTCDTMMManager`.

The XSLT Compiling processor detects the service provider for the `org.apache.xalan.xsltc.dom.XSLTCDTMMManager` service when a `javax.xml.transform.TransformerFactory` object is created. Any `javax.xml.transform.Transformer` or `javax.xml.transform.sax.TransformerHandler` objects that are created by using that `TransformerFactory` object will use the same

service provider. You can only 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 Tomcat, this limitation might apply.

If you are using an older version of Xerces or Xalan in the endorsed override, you might get a null pointer exception when you launch your application. This exception occurs because these older versions do not handle the jaxp.properties file correctly.

To avoid this situation, use one of the following workarounds:

- Upgrade to a newer version of the application that implements the latest Java API for XML Programming (JAXP) specification (<http://java.sun.com/xml/jaxp/index.html>).
- Remove the jaxp.properties file from /usr/lpp/java/J1.4_64/jre/lib
- Uncomment the entries in the jaxp.properties file in /usr/lpp/java/J1.4_64/jre/lib.
- Explicitly set the SAX Parser, DocumentBuilder, or Transformer factory using the IBM_JAVA_OPTIONS environment variable. For example:

```
set IBM_JAVA_OPTIONS=-Djavax.xml.parsers.SAXParserFactory=  
org.apache.xerces.jaxp.SAXParserFactoryImpl
```

or

```
set IBM_JAVA_OPTIONS=-Djavax.xml.parsers.DocumentBuilderFactory=  
org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
```

or

```
set IBM_JAVA_OPTIONS=-Djavax.xml.transform.TransformerFactory=  
org.apache.xalan.processor.TransformerFactoryImpl
```
- Set the system property for javax.xml.parsers.SAXParserFactory, javax.xml.parsers.DocumentBuilderFactory, or javax.xml.transform.TransformerFactory from within your application's code. For an example, see the JAXP 1.2 specification.

Chapter 14. Accessibility

The User Guides that are supplied with this SDK and the Runtime Environment have been tested by using screen readers. You can use a screen reader such as the Home Page Reader or the JAWS screen reader with these User Guides.

To change the font sizes in the User Guides, use the function that is supplied with your browser, usually found under the **View** menu option.

For users who require keyboard navigation, a description of useful keystrokes for Swing applications is in "Swing Component Keystroke Assignments" at <http://java.sun.com/j2se/1.4/docs/api/javawx/swing/doc-files/Key-Index.html>

Large Swing menu accessibility

If you have a Swing JMenu that has more entries than can be displayed on the screen, you can navigate through the menu items by using the down or up arrow keys.

Keyboard traversal of JComboBox components in Swing

If you traverse the drop-down list of a JComboBox component with the cursor keys, the button or editable field of the combo box does not change value until an item is selected. This is the desired behavior for this release and improves accessibility and usability by ensuring that the keyboard traversal behavior is consistent with mouse traversal behavior.

Chapter 15. Hints and tips

If you find a problem, see the "Hints and Tips" pages, at <http://www-1.ibm.com/servers/eserver/zseries/software/java/javafaq.html>.

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

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. Within the scope of the JVM, all character and string data is stored and manipulated in Unicode, and I/O data outside of the virtual machine (disk, network, and so forth) 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.

The Java language contains the abstractions necessary to handle the switch between character encodings. The various Reader and Writer classes in the `java.io` package provide alternate constructors with a specified codepage. This mechanism is used for internationalization support, and it can also be used to force ASCII (or other) I/O where required. Note that not all I/O needs to be overridden; for example, character output to the display should remain 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.

Some Java applications explicitly assume ASCII encoding and therefore require some alterations to run as expected on z/OS. For example, a platform-neutral application might have hard coded dependencies, such as literals in ASCII.

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

Support for IPv6

The class `java.net.NetworkInterface` does not currently support IPv6 addresses. The `java.net.NetworkInterface` methods `getNetworkInterfaces()`, `getByInetAddress()`, and `getByName()` return only network interfaces that have been configured with IPv4 addresses. The `java.net.NetworkInterface` method `getInetAddresses()` returns only IPv4 addresses.

Chapter 16. Reporting problems

If you find a problem that you have been unable to solve after looking through the "Hints and Tips" pages, see: <http://www-1.ibm.com/servers/eserver/zseries/software/java/services.html> for advice and information on how to raise problems.

Chapter 17. Notices

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

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

- IBM Director of Licensing
IBM Corporation
North Castle Drive, Armonk
NY 10504-1758 U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

- IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

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

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

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

Licensees of this program who wish to have information about it for the purpose of enabling (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

- JIMMAIL@uk.ibm.com
[Hursley Java Technology Center (JTC) contact]

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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

Trademarks

IBM, developerWorks, AS/400, z/OS and zSeries are trademarks or registered trademarks of International Business Machines Corporation in the United States, or other countries, or both.

Java and all Java-based trademarks are 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.

This product is also based in part on the work of the FreeType Project. For more information about FreeType, see <http://www.freetype.org>.

This product includes software developed by the Apache Software Foundation <http://www.apache.org/>.