IBM® Developer Kit for OS/390®, Java™ 2
Technology Edition

New IBM Technology featuring Persistent Reusable Java Virtual Machines

SC34-6034-01

IBM® Developer Kit for OS/390®, Java™ 2
Technology Edition

New IBM Technology featuring Persistent Reusable Java Virtual Machines

SC34-6034-01

Note

Before using this information and the product it supports, read the information in “Appendix D. Notices” on page 111.

Second Edition (October 2001)

This edition applies to the New IBM Technology featuring Persistent Reusable Java Virtual Machines, which is available as a special enhancement to the IBM Developer Kit for OS/390, Java 2 Technology Edition (product number 5655-D35), and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Contents

Figures

Tables

About this book

This book describes the external interface for the New IBM Technology featuring Persistent Reusable Java™ Virtual Machines. The OS/390 version of this technology, which is featured in this book, is available as a special enhancement to the IBM Developer Kit for OS/390, Java 2 Technology Edition.

This book also provides technical guidance on writing middleware and applications to run on the Persistent Reusable JVM.

This book comprises the following chapters:

- "Chapter 1. Introduction to the Persistent Reusable JVM" on page 1 gives an overview of the Persistent Reusable JVM, and introduces the concept of JVMSets.
- "Chapter 2. Developing a launcher subsystem" on page 7 describes launcher subsystems used to create and control Persistent Reusable JVMs and JVMSets.
- "Chapter 3. Performing system administration tasks" on page 57 describes system administrator tasks, and in particular the different types of class used by the Persistent Reusable JVM.
- "Chapter 4. Writing middleware" on page 61 explains how to write middleware for the Persistent Reusable JVM.
- "Chapter 5. Developing applications" on page 75 explains how to write applications for the Persistent Reusable JVM.
- "Chapter 6. Logging events" on page 81 describes how to log unresettable actions, reset trace events, and cross-heap events.
- "Chapter 7. Processing and debugging reset trace events" on page 85 describes why reset trace events occur, and provides information on how to fix the code to remove them.
- "Chapter 8. Using checked standard extensions" on page 89 describes how to load checked standard extensions that extend the core Java platform.
- "Appendix A. Unresettable reason codes" on page 91 lists the reason codes that detail why a Persistent Reusable JVM has been marked as unresettable.
- "Appendix B. Reset trace events" on page 95 lists the reset trace events that can occur during JVM-reset.
- "Appendix C. Application Programming Interface" on page 97 details the application programming interface (API) for the Persistent Reusable JVM. The API comprises hooks, JNI functions, Java classes, interfaces, and methods.
- "Glossary" on page 113 defines the terms used in this book.

Who should read this book

This book is aimed at:

- Transaction subsystem developers
- Middleware and application developers

It provides:

- A comprehensive overview of the Persistent Reusable JVM
- A detailed explanation of how to write a transaction subsystem launcher
- A guide to writing middleware and applications to run on the Persistent Reusable JVM

Conventions and terminology used in this book

Command line options, system parameters, and class names are shown in bold. For example:

- **-Xresettable**
- **-Xinitsh**
- **-Dibm.jvm.trusted.middleware.class.path**
- **java.security.SecureClassLoader**

Functions and methods are shown in a monospaced font. For example:

- `ResetJavaVM()`
- `QueryJavaVM()`

Options shown with values in braces signify that one of the values must be chosen. For example:

-Xverify:{*remote* | *all* | *none*}

Options shown with values in brackets signify that the values are optional. For example:

-Xrunhprof*[:help][:<suboption>=<value>,...]*

Prerequisite and related information

Books on related information are listed in “Bibliography” on page 115.

How to send your comments

Your feedback is important in helping to provide the most accurate and high-quality information. If you have any comments about this book or any other Java on OS/390 documentation, do one of the following:

- Go to the IBM **Java on the OS/390 Platform** home page at:

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

  There you will find the feedback page where you can enter and submit your comments.
- Send your comments by email to java@hursley.ibm.com. Include the name of the book, the part number of the book, the version of New IBM Technology featuring Persistent Reusable Java Virtual Machines, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

Summary of changes

Any changes made since the previous edition of the book are marked with vertical lines to the left of the changes. The technical changes included in this edition are:

- A JVMSet can now be created consisting of a master JVM and worker JVMs. See:
  - “Chapter 1. Introduction to the Persistent Reusable JVM” on page 1
  - Figure 3 on page 9, which shows a launcher subsystem for a JVMSet
  - A description of the new **-Xjvmset** options in “Nonstandard options” on page 14
  - “Configuring a JVMSet” on page 21
  - “Sample launcher for a JVMSet” on page 33.
- New garbage collection options: **-Xgcpolicy** and **-Xgcthreads**. See “Nonstandard options” on page 14.
- The **-Xoptionsfile** option, which allows Java options and environment settings to be passed by file. See “Nonstandard options” on page 14.
- A new `QueryGCStatus` function. See “Appendix C. Application Programming Interface” on page 97.
- A new `isResettableJVM` method. See “isResettableJVM method” on page 101.
- There are three hooks: abort, exit, and vfprintf. See “Standard options” on page 14 and “Appendix C. Application Programming Interface” on page 97.
- Event logging output can now be redirected to the vfprintf hook function. See the description of **-Dibm.jvm.events.output** in “System properties” on page 19.

Chapter 1. Introduction to the Persistent Reusable JVM

The New IBM Technology featuring Persistent Reusable Java Virtual Machines speeds up the processing of Java applications in transaction processing environments such as OS/390 systems. This book describes the external interface specification for this new technology, and features the OS/390 version of the Persistent Reusable Java Virtual Machine (JVM), which is available as a special enhancement to the IBM Developer Kit for OS/390, Java 2 Technology Edition.

Transaction processing in an OS/390 environment is characterized by short, repetitive transactions that run in subsystems such as Customer Information Control System (CICS®) Transaction Server or DB2® Database Management Systems. The Persistent Reusable JVM improves transaction processing throughput in such environments by providing the ability to:

- Run multiple JVMs within an OS/390 address space, thereby providing an element of scalability between transactions processed and JVMs used.
- Process hundreds or even thousands of transactions in a JVM which is reset between transactions or whenever necessary. This has the effect of distributing the cost of starting that JVM over all of the transactions processed by the JVM.

The new concepts that are used to improve transaction processing throughput are described in the following sections.

Sharing JVMs in an address space

To ensure isolation between transactions, each JVM processes only one transaction at a time, and each JVM is created in its own Language Environment (LE) enclave to ensure isolation between JVMs running in parallel. The set of JVMs within an address space is called a JVMSet. JVMs in a JVMSet all share a common system heap of system classes and other shareable classes. This reduces significantly the time needed to start a new JVM in the JVMSet because the majority of system classes will already be loaded in the system heap. It also reduces the overall memory footprint for these classes because they are loaded only once per JVMSet and not once per JVM.

A JVMSet comprises a master JVM and a set of (1-n) sharing worker JVMs. The master JVM controls the JVMSet in two ways:

- It provides the system heap which is shared by all the worker JVMs.
- It sets up the class-loading environment to be used to load primordial and shareable classes into the system heap.

Once created, the master JVM does not participate in any work. Its principle role is simply one of JVMSet initialization. Specifically, this involves the externalizing of its system heap to act as a class cache, and the externalizing of the JVM options that are relevant to the JVMSet, such as the class loading paths. This information is externalized by an encoded token which is returned from the master JVM creation call. The launching subsystem that created the master JVM is therefore able to pass this returned token as an option to each worker JVM when it is created.

There is a new JVM option for the Persistent Reusable JVM to control the creation of master and worker JVMs. This option is the **-Xjvmset** option.

Serial reusability

The model of one transaction per JVM implies the recycling of the JVM; that is, create a JVM, run the transaction, and destroy the JVM. However, the startup overheads for a traditional JVM are very high; clearly, high-volume transaction processing requires a model that allows serial reuse of a JVM by many transactions, and that destroys and creates a new JVM only when absolutely necessary.

A serially-reusable JVM is one that can be reset to a known state, normally between transactions; that is, after the application code has terminated for one transaction and before the application code starts for the next application. Once the JVM has been reset, the application code that runs in the next transaction is unable to determine whether it is running in a new JVM or a JVM that has been reset. As a result, the transaction cannot be affected by any actions of a previous transaction. This mode of operation is provided in the Persistent Reusable JVM by a JVM option, the **-Xresettable** option. When this option is specified, the JVM can be reset by a Java Native Interface (JNI) function, the `ResetJavaVM()` function. Providing the reset is successful, the following transaction can be executed in the same Persistent Reusable JVM; and providing the reset continues to be successful, it is possible to serially execute thousands of transactions in the same Persistent Reusable JVM. This serial reuse of the JVM improves transaction processing performance. The improved performance results from:

- Reduced JVM startup cost per transaction. The initial JVM startup cost is spread over the number of transactions that execute in the JVM, and the JVM-reset cost is relatively low per transaction.
- Reduced class-loading costs per transaction. This is because of the serial reusability of system classes, middleware classes, and shareable application classes.

If the JVM-reset fails, the launching system destroys the JVM and creates a new one. Whenever this occurs, transaction processing performance is degraded. So it is important to understand what causes the JVM-reset to fail. A reset-failure occurs when an action or condition makes the JVM unresettable. In the majority of cases this will be because an application class has performed an unresettable action.

Application classes

Application classes are not trusted by the Persistent Reusable JVM and must follow a strict set of rules to avoid making the JVM unresettable. For example, application classes must not:

- Modify global static variables, thereby changing the state of the JVM.
- Use the Abstract Windowing Toolkit (AWT) or any of its derivatives to access a display or keyboard, or to print, as this could change the state of the AWT which is treated as a nonresettable subsystem. For example, an application could find that the background screen color has been changed by the previous application.
- Directly load a native library, as it cannot be known what the native library provided by the application does. For example, it could modify static variables.
- Directly manage threads. This restriction exists to ensure that there cannot be any access to the transient heap while it is being cleared.

Such actions are not prevented but if used they potentially leave the JVM in an undefined state when the application is terminated. The application itself cannot be relied upon to restore this state, and the JVM is not allowed to restore state on behalf of the application as this would break the Java compliance rules. This is why such actions are unresettable actions, and why application code is treated by the JVM as untrustworthy. As a result, if an unresettable action is detected during the execution of the application, the JVM is marked unresettable and the launching subsystem destroys and re-creates the JVM.

The restrictions on application code are very similar (and in many cases identical) to the restrictions that have been defined for the serial execution of Enterprise JavaBeans™ (EJB) beans. See:

http://java.sun.com/products/ejb/docs.html

for the EJB Specification 1.1.

If application code is not trusted and therefore restricted in scope, how can you build functions (such as EJB bean Containers or Java interfaces) for systems such as CICS or MQSeries® which require the full scope of programming resources? The answer is to use trusted middleware.

Trusted middleware classes

Trusted middleware classes are loaded by a middleware class loader, which identifies the loaded classes to be trusted middleware. Middleware is trusted by the JVM to manage its own state across a JVM-reset. Middleware manages its own state in two parts:

- It tidies up its own state during the `ResetJavaVM()` function. This is accomplished using special Tidy-Up methods that can be defined for each middleware class. For example, typical tidy-up actions would be:
  - Release storage.
  - Terminate any threads that were created during the transaction.
  - Null out references from middleware objects to application objects. This ensures that the JVM-reset does not fail in garbage collection because of cross-heap references.

  As the middleware state can be controlled by Tidy-Up methods, it is possible for middleware to preserve state across resets of the JVM.
- It reinitializes its classes after a `ResetJavaVM()` call. It does this using special Reinitialize methods that can be defined for each middleware class, and which are executed on the first reuse of the class.

As a result, trusted middleware is allowed an unrestricted scope of operation.

Split heaps and heap-specific garbage collection

Split heaps provide a way of grouping objects by their expected lifetime. This grouping is then exploited by utilizing heap-specific, garbage-collection models. The heaps used are as follows:

System heap
: Contains class objects that persist for the lifetime of the JVM. These are:
  - System classes loaded by the bootstrap class loader. For example, java.lang.*.
  - Standard extension classes loaded by the extensions class loader.

- Middleware classes loaded by middleware class loaders which implement the Shareable interface. For example, the trusted middleware class loader (TMC) supplied with the Persistent Reusable JVM implements the Shareable interface.

These classes are loaded just once. Their static initializers are run just once. The system heap is never garbage collected.

Application-class system heap
This is a segregated part of the system heap, and contains shareable application class objects that persist for the lifetime of the JVM. Shareable application classes are loaded just once by application class loaders which implement the Shareable interface. For example, the shareable application class loader (SAC) supplied with the Persistent Reusable JVM implements the Shareable interface. Because these classes are reset during JVM-reset, this forces the static initializers to run on first usage. The application-class system heap is never garbage-collected.

Nonsystem heap
A fixed, preallocated area of contiguous memory that contains a middleware heap and a transient heap.

Middleware heap
Contains objects that have a life expectancy longer than a single transaction and that persist across JVM-resets. These include nonshareable class objects loaded by a middleware class loader supplied by a middleware vendor, and objects created in middleware context. During JVM-reset, Tidy-Up and Reinitialize methods can be used to reset the classes to a known initialization state ready for the next transaction. The middleware heap is subject to garbage collection, usually at JVM-reset time. The lifetime of these classes is controlled by the middleware.

Transient heap
Contains objects with a life expectancy tied to the transaction. These include classes loaded by the default application class loader (which loads classes using **-classpath**), plus any objects created in application context. Classes are loaded by the application code for each transaction. Their static initializers run on first usage. They are subject to garbage collection, although this is not generally required during the transaction. At JVM-reset, this heap is discarded and re-created (the size of the transient heap is specified with the **-Xinitth** option.). For the Persistent Reusable JVM to be resettable, there must be no *active* references from objects in the middleware heap to objects in the transient heap. If there are, the JVM is declared unresettable.

Figure 1 on page 5 shows the split-heap model and summarizes the actions that occur during a JVM-reset. Figure 4 on page 59 shows the relationship between class loaders and their target heaps.

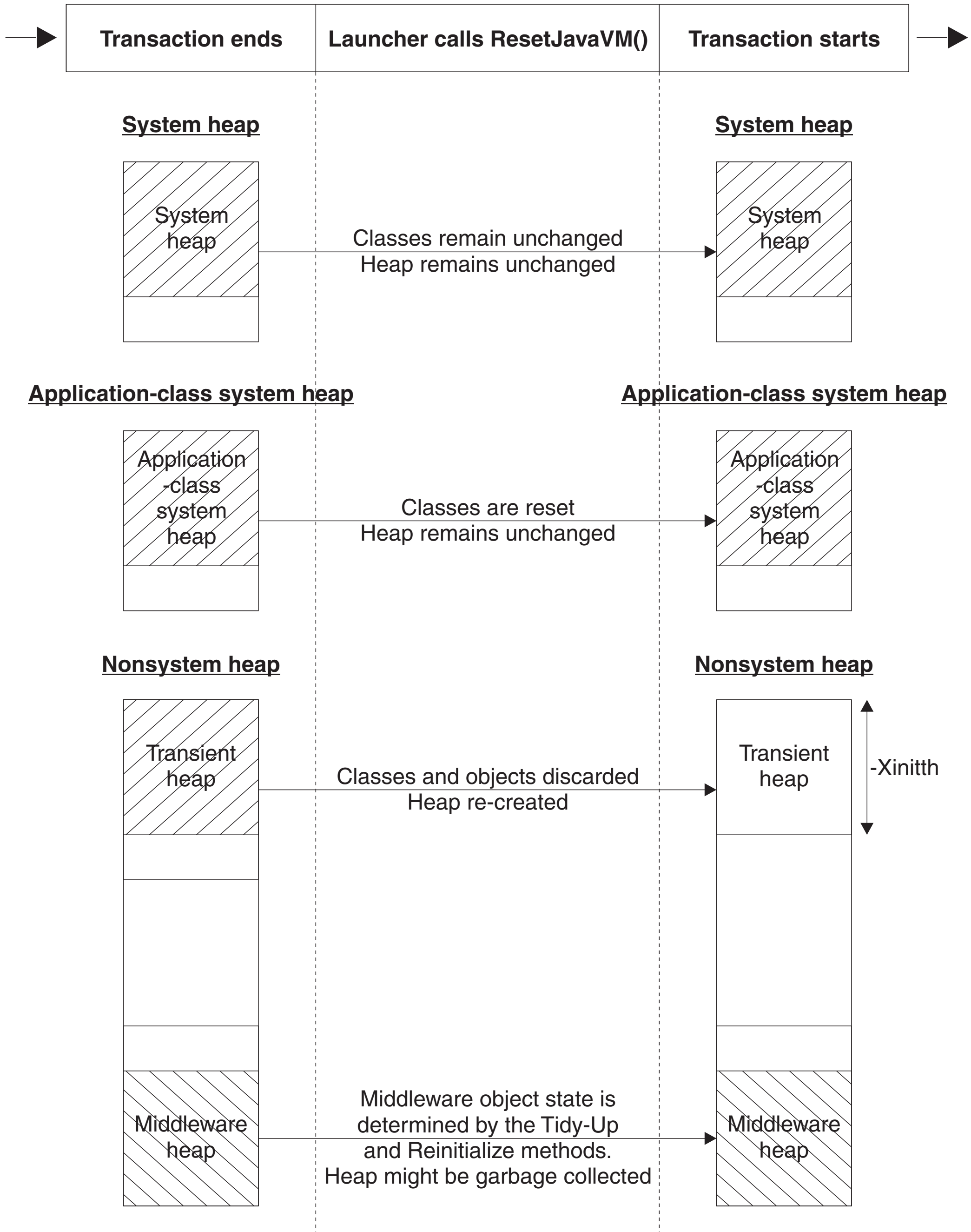

Figure 1. Split-heap model used in the Persistent Reusable JVM

Split heaps and heap-specific garbage collection

Chapter 2. Developing a launcher subsystem

This chapter:

- Gives an overview of launcher subsystems
- Explains how to start the debugging of an application
- Explains about prerequisites for Persistent Reusable JVMs
- Lists the command-line options, JVM options, and system properties
- Explains about Java command-line options
- Lists JVM option restrictions
- Gives the code for a sample launcher for a Persistent Reusable JVM
- Gives the code for a sample launcher for a JVMSet

Overview of launcher subsystems

To use the Persistent Reusable JVM for transaction processing, a subsystem must be developed to create and control the Persistent Reusable JVM, and to interface with a host transaction processing system, for example, CICS Transaction Server or DB2 products. This subsystem is referred to as a launcher, and the following sections review all aspects of the Persistent Reusable JVM that need to be addressed by a launcher subsystem. Figure 2 on page 8 illustrates a typical launcher subsystem for a reusable (standalone) JVM. Figure 3 on page 9 illustrates a typical launcher subsystem for a JVMSet.

JVM types

The launcher can launch either Persistent Reusable (standalone or JVMs in a JVMSet) or nonreusable JVMs. A Persistent Reusable JVM delivers the best performance for serial transactions. Nonreusable JVMs can run different types of transaction that cause the JVM to become unresettable.

A Persistent Reusable JVM uses a JNI function called `ResetJavaVM()`to reset its state at the end of a transaction, so that the next transaction sees a “new” JVM. For the launcher to be able to invoke the `ResetJavaVM()` function at the end of a transaction, it must launch the JVM initially using the `JNI_CreateJavaVM()` JNI function, so that it gains control when the application code associated with the transaction terminates. A simplified launcher example is shown in “Sample launcher for a Persistent Reusable JVM” on page 23. On OS/390 systems, the launcher might employ multiple LE enclaves to provide isolation.

Loading a class

Management of the different types of class that are used in the Persistent Reusable JVM, and how they are loaded by the JVM, is a system administration task (see “Loading classes” on page 57). To summarize, the Persistent Reusable JVM uses two class loaders to load shareable trusted middleware and shareable application classes. These class loaders are added to the class loader hierarchy in the Persistent Reusable JVM when the appropriate system properties (that define which class paths should be used by the class loaders) are specified at JVM creation time. See “Command-line options, JVM options, and system properties” on page 12.

Overview of launcher subsystems

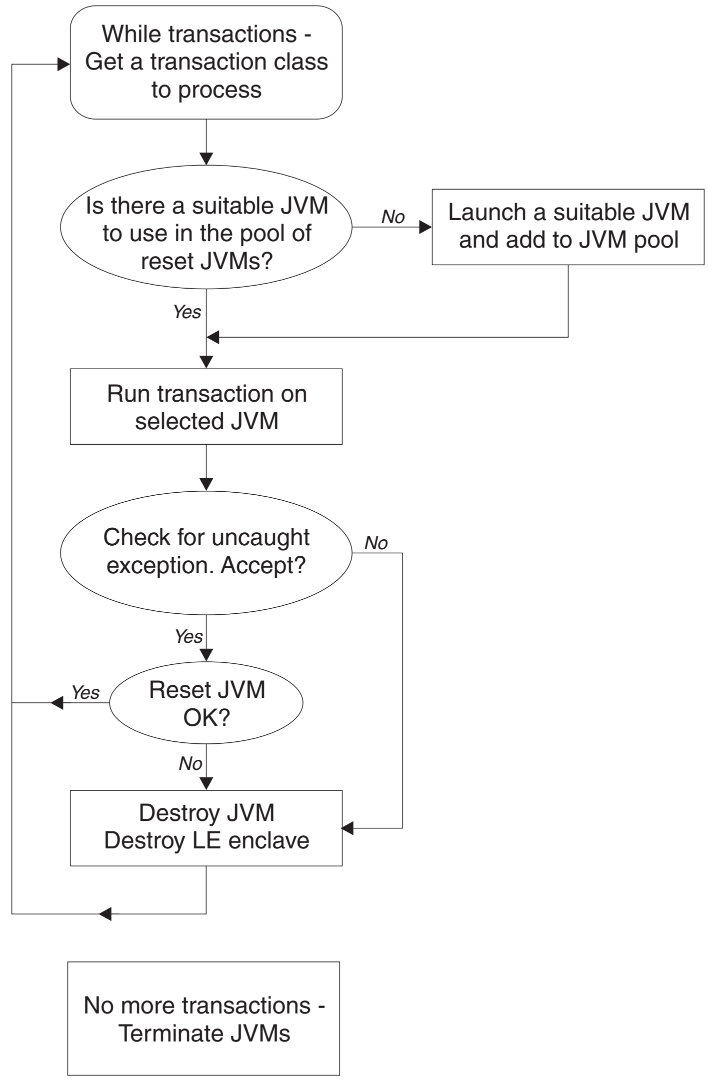

Figure 2. Launcher subsystem for a Persistent Reusable JVM

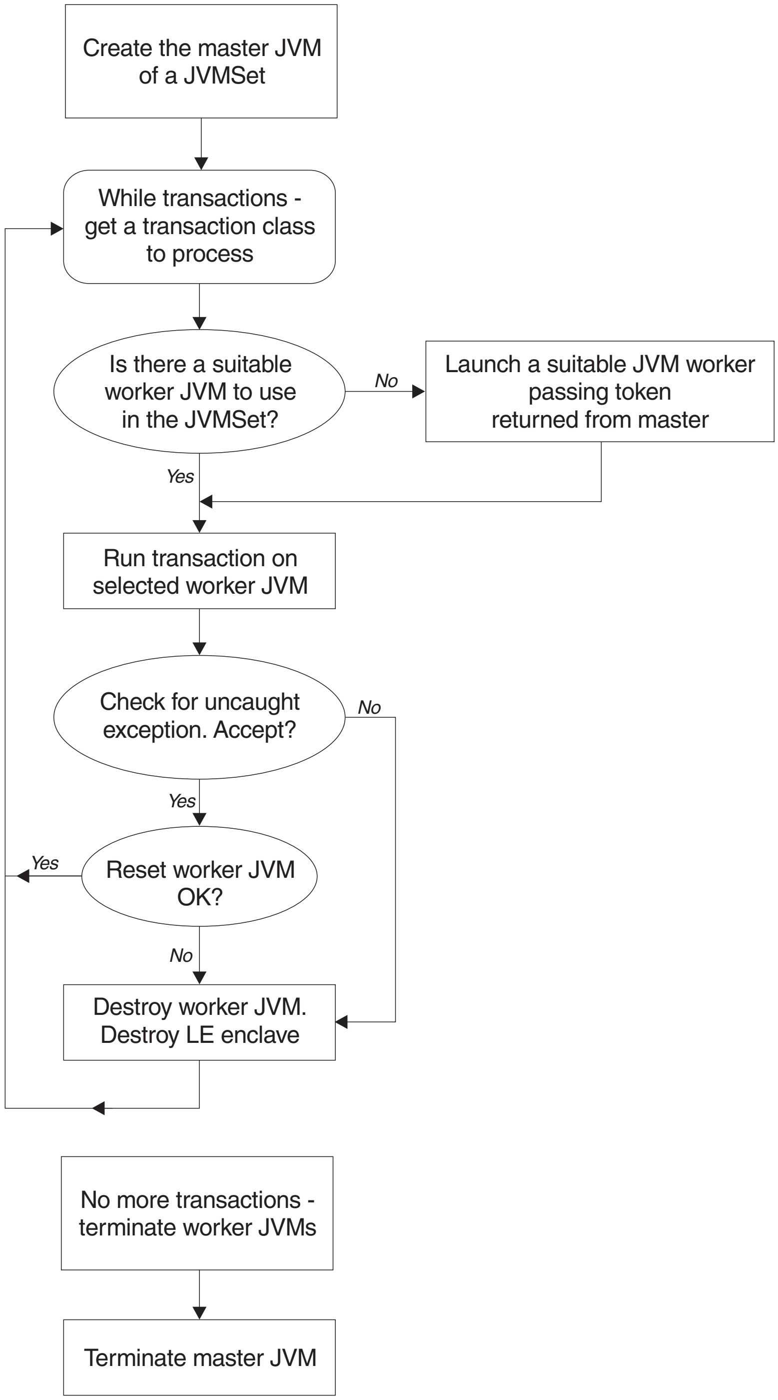

Figure 3. Launcher subsystem for a JVMSet

Choosing a JVM to use

The task of dispatching application code associated with a transaction on existing JVMs or newly-created JVMs lies with the transaction processing environment, for example, CICS or DB2. The choice of JVM to use from the pool already created (the pool of reusable JVMs or workers in a JVMSet) is based on such factors as the basic JVM options in force, such as resettability, system heap size, and class paths for middleware and shareable application class loaders.

If a suitable JVM does not exist, the launcher can create one with the necessary JVM options, which are defined in "Command-line options, JVM options, and system properties" on page 12. When a worker JVM is created, the worker is passed the token returned from the master JVM.

Running the application code for a transaction

Once a suitable JVM has been found or created, the launcher executes the necessary JNI code to load the application class associated with a transaction and create a class object, find the method ID to execute, and execute the method to run the transaction. In normal operation, when the application terminates, control is passed back to the launcher, at which point the launcher calls the `ResetJavaVM()` function. If the application does not terminate cleanly, and the launcher receives an uncaught Java exception which it is willing to accept, the launcher can still call `ResetJavaVM()`. If this is successful, the next transaction can be executed. Otherwise the JVM must be destroyed.

Handling abnormal conditions

If a JVM fails because of an uncaught exception, the launcher has two options. If the exception is acceptable (for example, a security failure), the launcher can clear the exception and call `ResetJavaVM()`. If the exception is not acceptable or unknown, the launcher must destroy the JVM.

Controlling access to JVM static variables

Because middleware is privileged to make changes to static data in the Persistent Reusable JVM without causing nonresettability, there is the concern that independently-written middleware components will make conflicting changes, especially during the tidy-up phase of JVM-reset. To avoid this, it is recommended that the launcher and its associated middleware code take responsibility for allowing access to JVM statics by providing getter/setter methods. By doing this, the launcher is in control and can maintain a consistent state for JVM static variables throughout the life of the JVM.

Resetting a JVM

When the application code for a transaction completes, the launcher decides whether to reset the JVM. Generally, it chooses to reset the JVM unconditionally. However, in some cases it might choose to query the state of the JVM first before making this decision. A JNI function (see "QueryJavaVM() JNI function" on page 108) can be used to determine the current state of the JVM, for example, to determine whether there are multiple threads running or whether there are application instance finalizers waiting to run. If the launcher decides to reset the JVM, it calls a JNI function (see "ResetJavaVM() JNI function" on page 109) which attempts to reset the JVM to a known initialization state. `ResetJavaVM()` calls any middleware Tidy-Up methods. The objective is to tidy up any transaction-related resources that were acquired, and to tidy up any references to application objects or application-class loader objects. These actions are necessary for a clean reset. For further details, see "Chapter 4. Writing middleware" on page 61.

Attention: The launcher must call `ResetJavaVM()` with the security context unchanged from that used by the application. This ensures:

- The correct context for application finalizers
- That middleware Tidy-Up methods run correctly

See “Using finalizers” on page 68.

If `ResetJavaVM()` returns JNI_OK, the reset was successful. A return of JNI_ERR indicates that the JVM is unresettable and is in an undetermined state. In this case, the launcher must destroy the JVM by calling `DestroyJavaVM()`. The launcher must also terminate the LE enclave so that normal system end-of-processing actions are performed and all traces of the JVM are removed. The reason why `ResetJavaVM()` returned JNI_ERR can only be determined by rerunning the application code for a transaction with logging turned on. See “Determining why a JVM becomes unresettable”. A return of JNI_EINVAL indicates that the JVM was not started with the **-Xresettable** option.

Required garbage collection

Every time the JVM is reset, the SAC and default nonshareable application class loaders are dereferenced (the references to the objects are removed). This leads to an accumulation of dereferenced objects in the nonsystem heap, and more importantly to an accumulation of system memory-allocated (malloc’d) storage that is associated with these objects. The more classes that were loaded by these class loaders, the larger this malloc’d storage is likely to be. The objects and malloc’d storage are not freed until the objects are garbage collected and the associated finalizer methods are run. A problem arises when the growth of the nonsystem heap is small, for example, when the nonsystem heap size is large but the growth in the associated system malloc’d memory is large (for example, transactions have loaded many classes). In this case, a garbage collection will not be triggered automatically by a heap-full condition even after many JVM-reset cycles, whereas the memory requirements reflected in working set size will be increasing. The launcher needs to be aware that garbage collection might be required periodically if the transaction mix has resulted in these conditions arising over a period of JVM-reset cycles.

Monitoring the state of heap growth is only possible by using the **-verbose:gc** JVM option.

Handling launcher shutdown

During launcher shutdown, JVMs created by the launcher can be terminated in any order, as required. Whenever a JVM is terminated, the associated LE enclave must also be terminated. For JVMSets, the worker JVMs can be terminated in any order and then the master JVM is terminated. The master JVM must be terminated last.

Determining why a JVM becomes unresettable

A logging facility has been implemented to assist in determining why a JVM becomes unresettable. Event logging is enabled using the following system property:

-Dibm.jvm.events.output=*{<path/filename>* **|** *stderr* **|** *stdout}*

which specifies where to log events. Unresettable event logging is enabled using the following system property:

-Dibm.jvm.unresettable.events.level=*{min* **|** *max}*

Overview of launcher subsystems

Logging of reset trace events is enabled using the following system property:

-Dibm.jvm.resettrace.events

Logging of cross-heap events is enabled using the following system property:

-Dibm.jvm.crossheap.events (This system property is active only when using a debug build.)

Note: These properties are read only during JVM startup.

If a JVM becomes unresettable while running a transaction, the only way to determine the reason is to enable these system properties and to rerun the transaction. It is good practice to check the application code for a transaction and middleware code prior to production use. For further details, see “Chapter 6. Logging events” on page 81. For further details on system properties, see “System properties” on page 19.

Sample launchers

“Sample launcher for a Persistent Reusable JVM” on page 23 illustrates a Persistent Reusable JVM being reused in a transaction processing loop. “Sample launcher for a JVMSet” on page 33 illustrates an example JVMSet launcher.

Debugging an application

To debug an application, the JVM that runs the application must be started with the **-Xdebug** option. This option can be added to any Persistent Reusable JVM. It is not necessary to explicitly disable the Just-in-Time (JIT) compiler because the JVM does not try to load it when debug is enabled. Using the debug option causes the debugging agent to be loaded, and this allows future connection of the debug tool using a password-protected socket connection.

Note: Specifying the **-Xdebug** option makes any JVM unresettable because the monitoring activity is performed on a separate thread that is asynchronous to the main Java thread that calls `ResetJavaVM()`. This could potentially interfere with system and programming resources during the call to `ResetJavaVM()`.

Prerequisites

No specific prerequisites are defined for running a Persistent Reusable JVM rather than a nonreusable JVM in an OS/390 system.

Command-line options, JVM options, and system properties

This section describes the command-line options, JVM options, and system properties for the Persistent Reusable JVM.

The Persistent Reusable JVM has a set of standard options that are supported in the OS/390 runtime environment. An additional set of nonstandard options are specific to the OS/390 virtual machine implementation, and are also supported by the Persistent Reusable JVM. Nonstandard options begin with **-X**. System properties begin with **-D**.

Command-line options

Command-line options are those options that are supplied to and interpreted by the default “java” launcher, as opposed to JVM options that any launcher can pass to a `JNI_CreateJavaVM()` call.

-classpath<*classpath*>
-cp<*classpath*>
: Specifies a list of directories, JAR archives, and ZIP archives to search for user class files. Class path entries are separated by a semicolon (;). Specifying **-classpath** or **-cp** overrides any setting of the CLASSPATH environment variable.

If **-classpath** and **-cp** are not used, and CLASSPATH is not set, the user class path consists of the current directory (.).

Note: When launching a JVM using the `JNI_CreateJavaVM()` function, the **-classpath** or **-cp** options are not passed to the JVM because they are “java launcher” options rather than JVM options. Also, the CLASSPATH environment variable will not be honoured because this is detected by the Java launcher and not by the JVM. When launching a JVM using `JNI_CreateJavaVM()`, the only way to define the classpath from which to load nonshareable application classes is via the **-Djava.class.path** system property.

-fullversion
: Displays version information (VM type, Java level, and build date) and exits. This does not include JIT information.

-jar<*filename*>
: Executes a program encapsulated in a JAR file. The first argument is the name of a JAR file instead of a startup class name. For this option to work, the manifest of the JAR file must contain a line of the form:

Main-Class: **classname**

where **classname** identifies the class having the:

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

method that serves as the starting point for the application.

When you use this option, the JAR file is the source of all user classes, and other user class path settings are ignored.

-noverify
: Turns verification off.

-showversion
: Displays version information (VM type, Java level, and build date), plus JIT information, and continues execution.

-verifyremote
: Runs the verifier on all code that is loaded into the system by a classloader.

-version
: Displays version information (VM type, Java level, build date, and JIT status) and exits. This includes information on the JIT compiler.

-Xfuture
: Performs strict class-file format checks. This flag turns on stricter class-file format checks that enforce closer conformance to the class-file format specification.

-?
-help Displays usage information and exits.

-X Displays information about nonstandard options and exits.

Standard options

-D_<propertyname=value>_
Defines _<propertyname>_ to equal _<value>_ in the system properties list.

-verbose:_{class_ **|** _Xclassdep_ **|** _gc_ **|** _jni}_
- _class_ - Displays information about each class loaded.
- _Xclassdep_ - Generates console output showing class dependency when loading classes.
- _gc_ - Reports on each garbage collection event.

  Note: -verbosegc can also be used.
- _jni_ - Reports information about the use of native methods and other JNI activity.

abort A hook for a function called when the JVM terminates abnormally. The “extraInfo” is a pointer to the hook function. See “Appendix C. Application Programming Interface” on page 97 for more information.

exit A hook for a function called when the JVM terminates normally. The ″extraInfo″ is a pointer to the hook function. See “Appendix C. Application Programming Interface” on page 97 for more information.

vfprintf
A hook for a function that redirects all VM messages. The ″extraInfo″ is a pointer to the hook function. See “Appendix C. Application Programming Interface” on page 97 for more information.

Nonstandard options

Specifying memory size

When specifying memory size in the following options:
- Append the letter k (uppercase or lowercase) to indicate KB[1]
- Append the letter m (uppercase or lowercase) to indicate MB[2]
- Append the letter g (uppercase or lowercase) to indicate GB[3]

-Xbootclasspath:_<bootclasspath>_
Specifies a colon-separated list of directories, JAR archives, and ZIP archives to search for boot class files. These are used in place of the boot class files included in the Java 2 SDK, Standard Edition software.

-Xcheck:_{jni_ **|** _nabounds}_
- _jni_ - Performs an additional check for JNI functions.
- _nabounds_ - Performs a native-array bounds check on JNI functions.

1. KB equals 1024 bytes.
2. MB equals 1 048 576 bytes (1024 x 1024 bytes)
3. GB equals 1 073 741 824 bytes (1024 x 1024 x 1024 bytes)

-Xdebug
Starts with the debugger enabled. The Java interpreter prints out a password for the use of the java debugger (jdb).

-Xgcpolicy:*{optthruput* | *optavgpause}*
Specifies the garbage collection behavior, making trade-offs between throughput of the application and overall system and the pause times caused by garbage collection.

- *optthruput* - 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 allocation requests can be satisfied quickly. Such garbage collection cycles introduce occasional unexpected pauses in the execution of application code. As applications grow in size and complexity, and heaps become correspondingly larger, this garbage collection pause time tends to grow in size and significance. The *optthruput* setting 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. The *optthruput* setting is the default setting.
- *optavgpause* - The *optavgpause* setting substantially reduces the time spent in garbage collection pauses, as well as limiting the effect of increasing heap size on the length of the garbage collection pause. This is particularly relevant to configurations with large heaps. (Consider a heap as large when it is at least 1 GB.) The pause times are reduced by overlapping garbage collection activities with normal program execution. This overlapping results in a small reduction to application throughput.

  The *optavgpause* setting is not allowed with the **-Xresettable** option.

If the Java heap becomes nearly full, and there is very little garbage to be reclaimed, requests for new objects might not be satisfied quickly because there is no space immediately available. If the heap is operated at near-full capacity, application performance might suffer regardless of which of the settings (*optthruput* or *optavgpause*) is used; and, if requests for more heap space continue to be made, the application receives an Out of Memory exception, which will result in JVM termination if the exception is not caught and handled. In these situations, you are recommended either to increase the heap size using the **-Xmx** parameter or to reduce the number of application objects in use.

For information on heap size tuning and the implications of garbage collection for application performance, see:

- http://www.ibm.com/developerworks/library/j-jtc/index.html
- http://www.ibm.com/developerworks/library/tip-heap-size.html
- http://www.ibm.com/developerworks/library/j-leaks/index.html

-Xgcthreads:*<n>*
Specifies the number of threads to be used for garbage collection. *<n>* must be at least 1 (the default is 1), otherwise an error message is issued and the JVM terminates. If this option is not specified, the total number of threads used for garbage collection is the same as the number of processors in the machine. This is the number used by the JVM.

-Xinitacsh*<size>*
Sets the initial size of the application-class system heap. In the Persistent

Reusable JVM, classes in this heap exist for the lifetime of the JVM. They are reset during a `ResetJavaVM()`, and so are serially reusable by applications running in the JVM. There is only one application-class system heap per Persistent Reusable JVM. In nonreusable mode, this option is ignored.

Example: **-Xinitacsh256k**
Default: 128 KB (OS/390 default)

-Xinitsh<*size*>
Sets the initial size of the system heap. In the Persistent Reusable JVM, classes in this heap exist for the lifetime of the JVM. They are unaffected by a `ResetJavaVM()`, and so are serially reusable by applications running in the JVM. The system heap is never subjected to garbage collection.

The maximum size of the system heap is unbounded.

Example: **-Xinitsh256k**
Default: 128 KB (OS/390 default)

-Xinitth<*size*>
In reusable mode, this sets the initial size of the transient heap within the nonsystem heap. If this is not specified and **-Xms** is, the initial size is taken to be half the **-Xms** value. If **-Xms** is not specified, a value of half the platform-dependent default value is used. In nonreusable mode, this option is ignored because there is no transient heap.

Example: **-Xinitth2M**
Default: 1000 KB/2 = 500 KB (OS/390 default)

-Xjvmset[*size*]
Creates a master JVM. An optional size in megabytes can be specified to set the total size of the shared memory segment. The default is 1MB.

When `JNI_CreateJavaVM()` returns successfully, the “extrainfo” field of the JavaVMOption contains the token to be passed to each worker. An attempt to create two master JVMs with the same token will fail.

The **-Xresettable** option must be used with this option when starting a master JVM.

Table 1 on page 21 lists the command-line options, JVM options, and system properties that can be used to configure a JVMSet.

-Xjvmset
Creates a worker JVM. The “extrainfo” field of the JavaVMOption must contain the token returned on the **-Xjvmset** option used to create the master JVM.

-Xmaxe<*size*>
In reusable mode, this option sets a maximum expansion size of <size>/2 for both the middleware and transient heaps.

In nonreusable mode there is no transient heap, so the value only applies to the middleware heap.

Example: **-Xmaxe4M**
Default: 0 (that is, there is no maximum value)

-Xmaxf_<size>_

Specifies the maximum percentage of free space for the middleware heap. This is a floating point number between 0 and 1, where 0 allows no free space in the heap. The middleware heap is shrunk if the free space exceeds this limit.

Example:	**-Xmaxf0.6**
Default:	0.6 (that is, 60%)(OS/390 default)

-Xmine_<size>_

In reusable mode, this option sets a minimum expansion size of <size>/2 for both the middleware and transient heaps.

In nonreusable mode there is no transient heap, so the value only applies to the middleware heap.

The default value if this is not specified is 1 MB.

Example:	**-Xmine2M**
Default:	1 MB (OS/390 default)

-Xminf_<size>_

In reusable mode, this option specifies the minimum percentage of free space for the middleware and transient heaps. This is a floating point number between 0 and 1, where 0 allows no free space in the heap. The heap grows if the free space is below the specified amount.

In nonreusable mode there is no transient heap, so the value only applies to the middleware heap.

If the option is not specified, the default value is 0.3, that is, 30%.

Example:	**-Xminf0.3**
Default:	0.3 (that is, 30%)(OS/390 default)

-Xms_<size>_

In reusable or nonreusable modes, this option sets the initial size of the middleware heap within the nonsystem heap.

If this option is not specified, a value of half the platform-dependent default value is used.

Example:	**-Xms10M**
Default:	1000 KB/2 = 500 KB (OS/390 default)

-Xmx_<size>_

In reusable mode, this option sets the maximum size of the nonsystem heap. The nonsystem heap is a contiguous region of virtual memory containing the middleware and transient heaps. The middleware heap grows from the bottom of this region, and the transient heap grows from the top of the region.

In nonreusable mode there is no transient heap, so the value of **-Xmx** sets the maximum middleware heap size.

If this option is not specified, a platform-dependent default value is used.

Example:	**-Xmx128M**
Default:	64 MB (OS/390 default)

-Xnoagent
Disables VM support for oldjdb.

-Xnoclassgc
Disables class garbage-collection.

-Xoptionsfile<*filename*>
Enables Java options and environment settings to be passed by file. The options file consists of a list of Java options encoded in the default platform encoding. Each option appears on a new line, and each line is terminated by a line terminator (`\n` or `\r` or `\r\n`). If the last character on the line is `\`, the next line is treated as a continuation of the current line - the `\` and the line terminator are discarded, and any leading white space characters on the continuation line are also discarded and are not part of the option. Any white space found between the option and the line terminator or line continuation marker is ignored. Also, any white space preceding the option is ignored. Blank lines are ignored. Comment lines are identified with a leading `#` symbol. Recursion is not allowed. If the **-Xoptionsfile** option is found when parsing an options file, an error is raised and the Java command is terminated.

An example options file is as follows:

```
# JVM startup options
# used with -Xoptionsfile Java option

# verbose options
  -verbo                  \
                 se:gc

# Garbage collection options
-mx96m
-Xgcthreads1
```

-Xoss<*size*>
Sets the size of the Java-code stack size for each thread. At least 1 KB must be specified.

Example: **-Xoss1M**
Default: 400 KB

-Xresettable
Enables resettability and allows the `ResetJavaVM()` function to be used to reset the JVM to a known state. If `ResetJavaVM()` is called and the JVM was not started with the **-Xresettable** option, `ResetJavaVM()` returns a failure code of JNI_EINVAL. Whenever this occurs the launching system must destroy the JVM.

This option must be used with the **-Xjvmset** option when starting a master JVM.

-Xrs Reduces the usage of operating system signals by the JVM.

-Xrun<*dllname*>*[:help][:*<*suboption*>*=*<*value*>*,...]*
Loads the given dynamic link library (dll) and passes it the list of options. For *dllname*, you can specify only hprof or jdwp (if you specify more than one of these options, it is the last-specified that is taken). These dlls drive JVMPI (to perform profiling) or JVMDI (to perform debugging).

-Xrunhprof*[:help][:*<*suboption*>*=*<*value*>*,...]*
Enables cpu, heap, or monitor profiling. This option is typically

followed by a list of comma-separated “<suboption>=<value>” pairs. Run the command **java -Xrunhprof:help** to obtain a list of suboptions and their default values.

For more information, see http://java.sun.com/j2se/1.3/docs/guide/jvmpi/.

-Xrunjdwp*[:help][:<suboption>=<value>,...]*
Loads the JPDA reference implementation in-process debugging libraries and passes any suboptions specified. This library resides in the target VM and uses the JVM debug interface (JVMDI) and the JNI to interact with it. It uses a transport and the Java Debug Wire Protocol (JDWP) to communicate with a separate debugger application.

For more information, see http://java.sun.com/products/jpda/doc/.

-Xservice=″<string>″
Takes a quoted string. String values allow a service engineer to modify JVM internal settings to help with analysis in the event of a reported problem.

Example: **-Xservice=″String provided by service engineer″**

-Xss*<size>*
Sets the size of the native code stack for each thread. At least 1 KB must be specified.

Example: **-Xss500k**
Default: 0.5 MB

-Xverify:*{remote* **|** *all* **|** *none}*
Specifies the level of verification the Persistent Reusable JVM uses when loading classes:

- *remote* - Verify only those classes that are loaded over the network. This is the default setting.
- *all* - Verify all classes.
- *none* - Do not perform verification.

System properties

-Dibm.jvm.crossheap.events
Enables the logging of cross-heap events. These are events generated because of cross-heap references from the middleware heap to the transient heap.

Notes:

1. To see these events, you must first enable general event logging as described in the **-Dibm.jvm.events.output** system property.
2. This system property is active only when using a debug build.

-Dibm.jvm.events.output=*{<path/filename>* **|** *stderr* **|** *stdout}*
Enables event logging in the JVM. It defines whether the text records describing the event are stored in a file described by its full path name, or whether the events are logged on the stderr or stdout print streams. The stderr and stdout options allow transaction processing systems such as CICS to merge the output from the streams, prefixing messages with date,

time, and system ID information. It is also possible to correlate (using the time stamp) the stack trace with a subsequent JVM-reset failure which identifies the userID.

If the destination is stdout or stderr and a vfprintf hook is defined, the output is redirected to the vfprintf hook function. The function receives a handle to the chosen destination (stdout or stderr) in the usual manner; this allows a user's code to route the output to the destination, or elsewhere, as required. If the destination is a file, the output goes to the file whether or not a vfprintf hook is defined. For more information on the vfprintf hook, see "Standard options" on page 14 and "Appendix C. Application Programming Interface" on page 97.

Example: **-Dibm.jvm.events.output=/u/pjr/javatest/log**
Default: None

-Dibm.jvm.resettrace.events

Enables the logging of reset trace events. These are events that have caused the JVM to perform a trace-for-unresettability check during garbage collection.

Note: To see these events, you must first enable general event logging as described in the **-Dibm.jvm.events.output** system property.

-Dibm.jvm.shareable.application.class.path=*<path>*

Defines where the shareable application class loader (SAC) looks for classes and Jar files. Use the ″:″ character to separate paths to Jar files and class directories that contain the application shared classes. This is the same syntax as used by CLASSPATH. If this property is not specified, the SAC is not created.

Example: **-Dibm.jvm.shareable.application.class.path=/u/pjr/app**
Default: None

-Dibm.jvm.trusted.middleware.class.path=*<path>*

Defines where the trusted middleware class loader (TMC) looks for classes and Jar files. Use the ″:″ character to separate paths to Jar files and class directories that contain the application shared classes. This is the same syntax as used by CLASSPATH. If this property is not specified, the TMC is not created.

Example: **-Dibm.jvm.trusted.middleware.class.path=/u/pjr/mw**
Default: None

-Dibm.jvm.unresettable.events.level=*{min* **|** *max}*

Enables the logging of unresettable events, and sets the level of logging required.

<min> - prints a list of reason codes that define the unresettable events found.

<max> - prints the reason codes plus a stack trace (where appropriate).

Note: To see these events, you must first enable general event logging as described in the **-Dibm.jvm.events.output** system property.

Example: **-Dibm.jvm.unresettable.events.level=max**
Default: None

Notes:

1. The checks for unresettable events stop once the JVM has been found to be unresettable, unless this property has been specified. When this property is specified, checking continues so that the application writer can discover all the reasons why the JVM became unresettable. Performance improves if this property is not set.
2. The only way to find out why a JVM becomes unresettable is to turn logging on. This is typically done once as part of acceptance trials for an application. For further details on the logging of unresettable events, see "Chapter 6. Logging events" on page 81.

-Djava.class.path=_<classpath>_
Specifies a list of directories, JAR archives, and ZIP archives to search for class files to be loaded by the default class loader, that is, the nonshareable application class loader. Class path entries are separated by a colon (:) on OS/390 systems.

Note: When launching a JVM using the `JNI_CreateJavaVM()` function, the only way of passing the default classpath to the JVM is via the **-Djava.class.path** system property. It is not possible to use the **-cp** or **-classpath** option, or the CLASSPATH environment variable, to set the class path for the default class loader unless the launching subsystem looks for these variables and sets **-Djava.class.path** accordingly.

-Djava.compiler=_<filename>_
Specifies the compiler to be used. Specify **-Djava.compiler=**_NONE_ to disable the loading of a compiler.

-Djava.security.policy=_<filename>_
Used to assign a policy file. A policy file specifies which permissions are available for code from various sources.

Configuring a JVMSet

The following table lists the command-line options, JVM options, and system properties that can be used to configure a JVMSet.

The key used in the table is as follows:

OK Option is valid.

Error Option will raise an error.

N/A Not applicable.

Table 1. Configuring a JVMSet

Command-line option, JVM option, or system property	Master	Worker	Standalone
-Dibm.jvm.events.output	OK	OK	OK
-Dibm.jvm.shareable.application.class.path	OK	Error	OK
-Dibm.jvm.trusted.middleware.class.path	OK	Error	OK
-Dibm.jvm.unresettable.events.level	OK	OK	OK
-Djava.compiler[6]	OK	OK	OK
-Djava.security.policy	OK	OK	OK

Table 1. Configuring a JVMSet (continued)

-noverify[3]	N/A	N/A	N/A
-verbose	OK	OK	OK
-verifyremote[3]	N/A	N/A	N/A
-Xbootclasspath	OK	Error	OK
-Xcheck	OK	OK	OK
-Xdebug[5]	Error	Error	OK
-Xfuture[3]	N/A	N/A	N/A
-Xgcthreads	OK	OK	OK
-Xinitacsh[4]	OK	Error	OK - ignored if not reusable
-Xinitsh[4]	OK	Error	OK
-Xinitth	OK	OK	OK - ignored if not reusable
-Xjvmset (token returned) [1]	OK	Error	N/A
-Xjvmset (token supplied) [1]	Error	OK	N/A
-Xmaxe, -Xmaxf, -Xmine, -Xminf, -Xms, -Xmx	OK	OK	OK
-Xnoclassgc	OK	OK	OK
-Xoss, -Xss	OK	OK	OK
-Xresettable[2, 4]	OK	Error	OK
-Xrs	OK	OK	OK
-Xrunhprof	OK	OK	OK
-Xverify	OK	OK	OK

Notes:

1. **-Xjvmset** is used to start both master and worker JVMs.
2. Only a master JVM can be started in resettable mode. See Note 4.
3. These options are used only by the default “java” launcher; they set the **-Xverify** option.
4. You cannot mix garbage collection models in a JVMSet. Each worker JVM inherits the garbage collection model used in the master JVM.
5. JVM debugging is not supported for shared classes because JVMDI replaces bytecodes to set breakpoints.
6. The value of “java.compiler” must be the same for each worker as for the master or “NONE”, that is, you can turn off the JIT compiler for a particular worker.

Java command-line options in the Persistent Reusable JVM

The Persistent Reusable JVM has been designed to be created and controlled from launcher code using the JNI. This allows the launcher to control the use of the `ResetJavaVM()` function which is only callable through the JNI. Consequently, although it would be possible to launch a JVM using the `java` command and specify any of the options listed, it would not be possible to call `ResetJavaVM()`.

JVM option restrictions

The following JVM options cannot be used in the Persistent Reusable JVM:

- **JAVA_ARGS**
- **-Xoldjava**
- **JNI_VERSION_1_1**

Sample launcher for a Persistent Reusable JVM

The launcher example demonstrates the following features:

- Setting up middleware and shareable application class paths.
- Creating a Persistent Reusable JVM with a set of specific options.
- Performing initial middleware setup, which includes creating a message file in memory from disk for use by subsequent applications.
- Executing a JVM-reset loop in which an application is identified (from a list of four) and launched by a middleware launcher. The use of Tidy-Up and Reinitialize methods is also shown.
- Accessing the middleware message file from the application to demonstrate the use of a middleware resource that spans JVM-resets.
- Performing an occasional call to garbage collection in the JVM-reset loop, as described in “Required garbage collection” on page 11.

For a **Hints and tips** forum on running Java in an OS/390 environment, see:

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

Directory structure

The example assumes that the following directory structure has been set up on OS/390 for user pjr, where /u/pjr/javasrc is a pointer to a local Persistent Reusable JVM implementation:

/u/pjr/javasrc-> /usr/lpp/java/IBM/bin

Once this has been set, the level of Java to be used can be checked by issuing the command:

java -fullversion

The root directory for the example is as follows:

/u/pjr/scjvm

Once created, copy the launcher.c code from the example to this directory.

Place all middleware and application java code in a /u/pjr/scjvm/source directory. This simplifies the compilation of this code as all the classes are local to the directory. Once compiled, the middleware class files must be placed in a /u/pjr/scjvm/mw directory, and the application class files must be placed in a /u/pjr/scjvm/app directory. This ensures that these classes will be loaded by the appropriate class loader.

Finally, place the file “MessageFile” in the /u/pjr/scjvm directory.

Running the launcher example

The launcher code has been set up for cross-platform usage. To execute on the OS/390 platform, pass the USE_390 flag as a debug argument when compiling launcher.c.

launcher.c compilation

The following command script can be used to compile and link launcher.c. Create this script as a command called, for example, mkc.

```
#!
c++ -c -DUSE_390 -W "0,langlvl(extended)" -W c,ss -W c,dll -W
"c,float(ieee)" -I /u/pjr/javasrc/../include -o $1.o $1.c
c++ -o$1 -W l,dll $1.o /u/pjr/javasrc/../bin/classic/libjvm.x
```

To compile and link launcher.c, issue the commands:

cd /u/pjr/scjvm
mkc launcher

Java compilation

Compile the middleware and application java code in the usual way. For example:

cd /u/pjr/scjvm/source
javac MWMessageLoader.java
javac MWLaunchApp.java
javac App0.java
javac App1.java
javac App2.java
javac app3.java

Now move the class files to the appropriate class loading directory:

mv MW*.class ../mw
mv App*.class ../app

Runtime setup

export LIBPATH=/u/pjr/javasrc/bin/classic:$LIBPATH
export LIBPATH=/u/pjr/javasrc/bin:$LIBPATH
export LIBPATH=/usr/lib:$LIBPATH

Execution

The launcher example can be executed as follows:

cd /u/pjr/scjvm
launcher

Launcher code (launcher.c)

```
#include <stdio.h>
#include <jni.h>
#ifdef USE_390
  #include <unistd.h>
#endif
typedef unsigned short bool_t;
#define FALSE 0
#define TRUE  1

int main(int argc, char *argv[])
{
  char classpath[]     = "-Djava.class.path=.";
#if defined(USE_390) || defined(USE_AIX)
```

```
  char MWclasspath[]   = "-Dibm.jvm.trusted.middleware.class.path=/u/pjr/scjvm/mw";
  char Appclasspath[]  = "-Dibm.jvm.shareable.application.class.path=/u/pjr/scjvm/app";
#else
  char MWclasspath[]   =
           "-Dibm.jvm.trusted.middleware.class.path=f:\\notebook\\java\\scjvm\\mw";
  char Appclasspath[]  =
           "-Dibm.jvm.shareable.application.class.path=f:\\notebook\\java\\scjvm\\app";
#endif
  char Xresettable[]   = "-Xresettable";
  char Verbosegc[]     = "-verbose:gc";
  char logname[]       = "-Dibm.jvm.events.output=f:\\notebook\\java\\scjvm\\log";
  char URevents[]      = "-Dibm.jvm.unresettable.events.level=max";
  char RTevents[]      = "-Dibm.jvm.resettrace.events";

  char javaSystem[]    = "java/lang/System";

  char mainStr[]       = "main";
  char initStr[]       = "<init>";
  char gcStr[]         = "gc";
  char sigStringVoid[] = "(Ljava/lang/String;)V";
  char sigVoid[]       = "()V";

  char * applic[4]     = {"App0", "App1", "App2", "App3"};
  int AppCount         = 4;

  char mwMessageLoaderStr[] = "MWMessageLoader";
  char MessageFileStr[]     = "MessageFile";
  char mwLaunchAppStr[]     = "MWLaunchApp";

  bool_t         resettable   = TRUE;
  bool_t         resetlog     = TRUE;
  bool_t         verbosegc    = FALSE;
  bool_t         resetFailed  = FALSE;
  int            resetCount   = 0;
  int            gcCount      = 0;
  int            iter         = AppCount;   /* set number of ResetJavaVM iterations */

  JNIEnv        *env;
  JavaVM        *jvm;
  JavaVMInitArgs jvm_args;
  JavaVMOption   options[30];
  jint           rc;

  jclass         systemCid;
  jmethodID      gcMid;
  jclass         strCid;
  jstring        AppJStr;
  jclass         mwMessageLoaderCid;
  jobject        mwMessageLoaderObj;
  jmethodID      mwMessageLoaderMid;
  jclass         mwLaunchAppCid;
  jmethodID      mwLaunchAppMid;
  jstring        MessageFileJStr;

  int            i;


#ifdef USE_390
  __etoa(classpath    );
  __etoa(MWclasspath  );
  __etoa(Appclasspath );
  __etoa(Xresettable  );
  __etoa(logname      );
  __etoa(URevents     );
  __etoa(RTevents     );
  __etoa(Verbosegc    );
  __etoa(javaSystem   );
  __etoa(mainStr      );
  __etoa(initStr      );
  __etoa(gcStr        );
  __etoa(sigStringVoid);
  __etoa(sigVoid      );
```

```
  __etoa(mwMessageLoaderStr);
  __etoa(MessageFileStr);
  __etoa(MWLaunchAppStr);
  for (i=0; i<AppCount; i++)
  {
    __etoa(applic[i]);
  }
#endif

  options[0].optionString = classpath;
  options[1].optionString = MWclasspath;
  options[2].optionString = Appclasspath;

  i=3;
  /* Optional settings follow */
  if (resettable)
  {
    options[i].optionString = Xresettable;
    i++;
    printf("about to create a Resettable JVM...\n");
  }
  else
  {
    printf("about to create a Nonresettable JVM...\n");
  }
  if (resetlog)
  {
    printf("with Logging...\n");
    options[i].optionString = logname;
    i++;
    options[i].optionString = URevents;
    i++;
    options[i].optionString = RTevents;
    i++;
  }
  if (verbosegc)
  {
    printf("with Verbose:gc enabled...\n");
options[i].optionString = Verbosegc;
    i++;
  }
  /* Mandatory settings follow */
  jvm_args.version = 0x00010002;
  jvm_args.options = options;
  jvm_args.nOptions = i;

  /*
   * Create the jvm
   */
  rc = JNI_CreateJavaVM(&jvm, (void **)&env, &jvm_args);

  printf("JNI_CreateJavaVM rc = %i\n",rc);
  if (rc != 0)
  {
    printf("**error  JVM failed to create\n");
    exit (-1);
  }
  /*
   * Get the gc method id for subsequent use
   */
  systemCid = (*env)->FindClass(env, javaSystem);
  gcMid = (*env)->GetStaticMethodID(env, systemCid, gcStr, sigVoid);
  /*
   * Load and run the Middleware class MWMessageLoader to load
   * the message file
   */
  mwMessageLoaderCid = (*env)->FindClass(env, mwMessageLoaderStr);
  if (mwMessageLoaderCid == 0)
  {
    printf("**error Failed to find MWMessageLoader class\n");
    exit (-2);
  }
```

```
mwMessageLoaderMid = (*env)->GetMethodID(env, mwMessageLoaderCid, initStr, sigStringVoid);
if (mwMessageLoaderMid == 0)
{
  printf("**error Failed to find MWMessageLoader constructor\n");
  exit (-3);
}
/*
 * Construct a java string for MessageFileStr and run the
 * constructor for MWMessageLoader
 */
MessageFileJStr = (*env)->NewStringUTF(env, MessageFileStr);

mwMessageLoaderObj = (*env)->NewObject(env, mwMessageLoaderCid,
  mwMessageLoaderMid, MessageFileJStr);
if (!mwMessageLoaderObj)
{
  printf("error Failed to load Message File\n");
exit (-4);
}
/*
 * Load and get the MWLaunchApp main method id for subsequent use
 */
mwLaunchAppCid = (*env)->FindClass(env, mwLaunchAppStr);
if (mwLaunchAppCid == 0)
{
  printf("**error Cannot find MWLaunchApp class\n");
  exit (-5);
}
mwLaunchAppMid = (*env)->GetStaticMethodID(env, mwLaunchAppCid, mainStr, sigStringVoid);
if (mwLaunchAppMid == 0)
{
  printf("error  Cannot find MWLaunchApp main method\n");
  exit (-6);
}

/**************** ResetJavaVM loop starts here *******************/
while (!resetFailed)
{
  /**
   * Run a series of Applications - the limit is set by the
   * variable iter.For the purposes of this example we assume that
   * each application will be launched by the middleware
   * class MWLaunchApp.  Application names are looked up in the
   * array applic[] and are passed to the main method of the
   * MWLaunchApp class.
   **/

 /*
  * Convert the UTF8 name for the application into a JString
  */
  AppJStr = (*env)->NewStringUTF(env, applic[resetCount]);
  if (AppJStr == 0) {
    printf("Could not construct a JString for the Application name\n");
    exit(-7);
  }
  /*
   * Call the main method of MWLaunchApp and pass the application
   * name to launch
   */
  (*env)->CallStaticVoidMethod(env, mwLaunchAppCid, mwLaunchAppMid, AppJStr);

 /*
  * Transaction has finished - now reset the JVM
  */
  rc = (*jvm)->ResetJavaVM(jvm);
  resetCount++;
  gcCount++;
  printf("\nLauncher: ResetJavaVM returned rc = %i, resetCount = %i\n\n",rc,resetCount);
  if (rc != 0)
  {
    resetFailed = TRUE;
  }
```

Sample launcher for a Persistent Reusable JVM

```
    else if (resetCount >= iter)
    {
      printf("Launcher: %i applications have been processed - reset
          loop will now terminate\n", resetCount);
      resetFailed = TRUE;
    }
    else if (gcCount == 3)
    {
   /*
    * This demonstrates how to call gc at regular intervals as
    * recommended in the 'Required garbage collection' section.
    * The value 3 here has no significance.
    */
      printf("Launcher: gcCount = %i so invoking a GC....\n",gcCount);
      (*env)->CallStaticVoidMethod(env, systemCid, gcMid);
      gcCount = 0;
    }
  }
  /***************  ResetJavaVM loop ends here ********************/

  rc = (*jvm)->DestroyJavaVM(jvm);
  printf("DestroyJavaVM returned rc = %i\n",rc);

  return 0;
}
```

Middleware code

MWMessageLoader class

```
/**
 * Middleware class to read a file of text message records into a
 * static String array. These can be subsequently
 * accessed by applications to demonstrate the use of a middleware
 * controlled resource that maintains its state
 * across JVM-resets.  This class does not therefore require any
 * Tidy-Up or Reinitialize methods.
 **/
import java.io.*;

public class MWMessageLoader {
  private static File f;
  private static BufferedReader br;
  private static FileReader fr;

  public static String Messages[] = new String[30];
  public static int MessageFileSize;

  static {
      /* print which class loader loaded me */
      try {
      System.out.println("Class MWMessageLoader was loaded by " +
      Class.forName("MWMessageLoader").getClassLoader().toString());
      }
      catch (Exception e) {
      }
  }

  /**
   * Constructor to initialise the Message Buffer
   **/
  public MWMessageLoader(String MessageFileStr) {

    int i;

    f = new File(MessageFileStr);
    try {
      fr = new FileReader(f);
      br = new BufferedReader(fr);

      for (i = 0; i<1000; i++) {
        try {
```

```
Messages[i] = br.readLine();
if (Messages[i].length() <= 0) {
  MessageFileSize = i;
  break;
}
      }
      catch (IOException e) {
        System.out.println(e.getClass().getName() + ":" + e.getMessage());
      }
    }
    System.out.println("MWMessageLoader: Message File has been loaded");
    for (i = 0; i<MessageFileSize; i++) {
      System.out.println("Message( " + i + ")" + Messages[i]);
    }
  }
  catch (FileNotFoundException e) {
    System.out.println(e.getClass().getName() + ":" + e.getMessage());
  }
```

MWLaunchApp class

```
/**
 * Middleware class to launch an application and demonstrate the use
 * of Reinitialize and Tidy-up methods.
 **/
 import java.lang.reflect.*;
 import com.ibm.jvm.ExtendedSystem;

 public class MWLaunchApp extends Object
 {
   static ClassLoader contextCL;

   /**
    * static initializer
    */
   static
   {
       /* print which class loader loaded me */
       try {
       System.out.println("Class MWLaunchApp was loaded by " +
       Class.forName("MWLaunchApp").getClassLoader().toString());
       }
       catch (Exception e) {
       }

       ibmJVMReinitialize();
   }

   /**
    * Reinitialize method
    */
   private static void ibmJVMReinitialize()
   {
   /*
    * Re-establish the context class loader on each use of this class
    */
     System.out.println("MWLaunchApp.Reinitialize: re-establishing the context
       class loader");
     contextCL = Thread.currentThread().getContextClassLoader();
   }

   /**
    * Tidy-Up method
    */
   private static boolean ibmJVMTidyUp()
   {
   /*
    * Before JVM-reset we must perform any tidy-up actions for this
    * class. For efficiency this can be bypassed if the JVM is already
    * marked unresettable. For this simplified example  the tidy up
    * actions are to dereference the context class loader reference
    */
```

```
      if (!ExtendedSystem.isJVMUnresettable())
      {
        System.out.println("MWLaunchApp.TidyUp: dereferencing middleware
            application references");
        contextCL = null;
      }
      else
        System.out.println("MWLaunchApp.TidyUp: skipping tidy-up actions
            as JVM is unresettable");

      return true;
    }

    /**
     * Main method - load and run the application named in the input
     * parameter.  For simplicity we assume
     * the method to execute is called appMethod with no parameters.
     **/
    public static void main(String appName)
    {
      try
      {
        Class AppClassRef = contextCL.loadClass(appName);
                                              /* Load the class  */
        Object AppClassObj = AppClassRef.newInstance();
                                 /* Get a new instance of class  */
        Method AppMethod = AppClassRef.getMethod("appMethod",null);
                                  /* Get the appMethod method id */
        AppMethod.invoke(AppClassObj,null);
        AppClassRef = null;
        AppClassObj - null;
                                    /* Execute the method */
      }
      catch (Exception e)
      {
        System.out.println("** error The application (" + appName + ") did not execute");
        System.out.println("Exception was: " + e);
      }
    }
  }
```

Application code

For brevity, just one of the dummy applications is shown in the following. To run the launcher code, construct similar application code for App1.java, App2.java, and App3.java.

```
import MWMessageLoader;

/**
 * Dummy Application class App0
 **/
public class App0 extends Object
{
  static
  {
    System.out.println("App0 static initializer");
    try {
    System.out.println("Class App0 was loaded by " +
       Class.forName("App0").getClassLoader().toString());
    }
    catch (Exception e) {
    }
  }

  /**
   * Constructor
   */
  public App0()
  {
    System.out.println("App0 constructor");
```

```
  }

  /**
   * Application method which we assume takes no parameters and does
   * all the work for the transaction
   **/
  public void appMethod()
  {
    /*
     * As an example - lookup message 0 in  MessageFile
     */

    System.out.println("App0.appMethod: " + MWMessageLoader.Messages[0] );
  }
}
```

Example of logging output

When the launcher executes, logging is enabled and a file called log will be created in the /u/pjr/scjvm directory. The log file produced by this launcher example is as follows:

```
[****************** EVENT LOG FILE HEADER *****************]
START DATE: Thu Feb 22 14:03:29 2001
MILLIS    : 657
******************* SYSTEM PROPERTIES **********************
java.assistive=ON
java.runtime.name=Java(TM) 2 Runtime Environment, Standard Edition
sun.boot.library.path=f:\jdk1.3s\inst.images\x86_nt_4\sdk\jre\bin
java.vm.version=1.3.0
java.vm.vendor=IBM Corporation
java.vendor.url=http://www.ibm.com/
path.separator=;
java.vm.name=Classic VM
file.encoding.pkg=sun.io
java.vm.specification.name=Java Virtual Machine Specification
user.dir=F:\NOTEBOOK\java\scjvm
java.runtime.version=1.3.0
java.fullversion=J2RE 1.3.0 IBM Windows 32 build hndev-20010207 (JIT disabled)
java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment
os.arch=x86
java.io.tmpdir=C:\DOCUME~1\pjr\LOCALS~1\Temp\
line.separator=

java.vm.specification.vendor=Sun Microsystems Inc.
java.awt.fonts=
os.name=Windows 2000
java.library.path=F:\NOTEBOOK\java\scjvm;.;C:\WINNT\System32;
C:\WINNT;f:\jdk1.3s\inst.images\x86_nt_4\sdk\jre\bin;
f:\jdk1.3s\inst.images\x86_nt_4\sdk\bin;
f:\jdk1.3s\inst.images\x86_nt_4\sdk\jre\bin\classic;
C:\WINNT\SYSTEM32;C:\WINNT;C:\WINNT\System32\Wbem;
C:\Lotus\Notes;F:\CMVC\CMVC\EXE;F:\CMDS;
c:\program files\devstudio\sharedide\bin\ide;
c:\program files\devstudio\sharedide\bin;
c:\program files\devstudio\vc\bin
ibm.jvm.events.output=f:\notebook\java\scjvm\log
ibm.jvm.shareable.application.class.path=f:\notebook\java\scjvm\app
java.specification.name=Java Platform API Specification
java.class.version=46.0
ibm.jvm.unresettable.events.level=max
os.version=5.0
user.home=C:\Documents and Settings\pjr
user.timezone=
java.awt.printerjob=sun.awt.windows.WPrinterJob
file.encoding=Cp1252
java.specification.version=1.2
user.name=pjr
java.class.path=.
java.vm.specification.version=1.0
java.home=f:\jdk1.3s\inst.images\x86_nt_4\sdk\jre
```

```
java.specification.vendor=Sun Microsystems Inc.
user.language=en
awt.toolkit=sun.awt.windows.WToolkit
java.vm.info=J2RE 1.3.0 IBM Windows 32 build hndev-20010207 (JIT disabled)
java.version=1.3.0
java.ext.dirs=f:\jdk1.3s\inst.images\x86_nt_4\sdk\jre\lib\ext
sun.boot.class.path=f:\jdk1.3s\inst.images\x86_nt_4\sdk\jre\lib\rt.jar;
f:\jdk1.3s\inst.images\x86_nt_4\sdk\jre\lib\i18n.jar;
f:\jdk1.3s\inst.images\x86_nt_4\sdk\jre\classes
java.vendor=IBM Corporation
file.separator=\
java.vendor.url.bug=
java.compiler=none
sun.io.unicode.encoding=UnicodeLittle
ibm.jvm.trusted.middleware.class.path=f:\notebook\java\scjvm\mw
ibm.jvm.resettrace.events=
user.region=GB
****************END SYSTEM PROPERTIES **********************
[****************** END EVENT FILE HEADER *****************]
[EVENT 0x1]
TIME=22/02/2001 at 14:03:30.418
THREAD=main (0:2f5770)
CLASS=ResetJVMEvent
DESCRIPTION=JVM reset number 0x1 completed successfully
[END EVENT]
[EVENT 0x1]
TIME=22/02/2001 at 14:03:30.519
THREAD=main (0:2f5770)
CLASS=ResetJVMEvent
DESCRIPTION=JVM reset number 0x2 completed successfully
[END EVENT]
[EVENT 0x1]
TIME=22/02/2001 at 14:03:30.609
THREAD=main (0:2f5770)
CLASS=ResetJVMEvent
DESCRIPTION=JVM reset number 0x3 completed successfully
[END EVENT]
[EVENT 0x1]
TIME=22/02/2001 at 14:03:30.739
THREAD=main (0:2f5770)
CLASS=ResetJVMEvent
DESCRIPTION=JVM reset number 0x4 completed successfully
[END EVENT]
```

Example of online output

The output produced on stdout when the launcher runs is as follows:

```
F:\NOTEBOOK\java\scjvm>launcher
about to create a Resettable JVM...
with Logging...
JVMDG200: Diagnostics system property ibm.jvm.events.output=f:\notebook\java\scj
vm\log
Warning: JIT compiler "none" not found. Will use interpreter.
JNI_CreateJavaVM rc = 0
Class MWMessageLoader was loaded by sun.misc.Launcher$MiddlewareClassLoader@2668
0ae4
MWMessageLoader: Message File has been loaded
Message( 0)-----------------This is message number 0 -----------------
Message( 1)-----------------This is message number 1 -----------------
Message( 2)-----------------This is message number 2 -----------------
Message( 3)-----------------This is message number 3 -----------------
Message( 4)-----------------This is message number 4 -----------------
Message( 5)-----------------This is message number 5 -----------------
Class MWLaunchApp was loaded by sun.misc.Launcher$MiddlewareClassLoader@26680ae4

MWLaunchApp.Reinitialize: restablishing the context class loader
App0 static initializer
Class App0 was loaded by sun.misc.Launcher$ShareableClassLoader@26774ae4
App0 constructor
App0.appMethod: -----------------This is message number 0 -----------------
MWLaunchApp.TidyUp: dereferencing middleware application references
```

```
Launcher: ResetJavaVM returned rc = 0, resetCount = 1

MWLaunchApp.Reinitialize: re-establishing the context class loader
App1 static initializer
Class App1 was loaded by sun.misc.Launcher$ShareableClassLoader@267e8ae4
App1 constructor
App1 appMethod
App1.appMethod: -----------------This is message number 1 -----------------
MWLaunchApp.TidyUp: dereferencing middleware application references

Launcher: ResetJavaVM returned rc = 0, resetCount = 2

MWLaunchApp.Reinitialize: re-establishing the context class loader
App2 static initializer
Class App2 was loaded by sun.misc.Launcher$ShareableClassLoader@26780ae4
App2 constructor
App2.appMethod: -----------------This is message number 2 -----------------
MWLaunchApp.TidyUp: dereferencing middleware application references

Launcher: ResetJavaVM returned rc = 0, resetCount = 3

Launcher: gcCount = 3 so invoking a GC....
MWLaunchApp.Reinitialize: re-establishing the context class loader
App3 static initializer
Class App3 was loaded by sun.misc.Launcher$ShareableClassLoader@26458ae4
App3 constructor
App3.appMethod: -----------------This is message number 3 -----------------
MWLaunchApp.TidyUp: dereferencing middleware application references

Launcher: ResetJavaVM returned rc = 0, resetCount = 4

Launcher: 4 applications have been processed - reset loop will now terminate
DestroyJavaVM returned rc = -1

F:\NOTEBOOK\java\scjvm>
```

Sample launcher for a JVMSet

The following is a sample launcher for a JVMSet. The sample consists of the following files:

- go.c (launcher and monitor program)
- LauncherHeader.h (Header file including typedefs and prototypes for functions)
- LauncherFuncs.c (implementations of the functions)
- jvmcreate.c (creates a JVM and requests and runs tasks on it)
- go.prp (sample properties file)
- testclasses.txt (sample task file)
- HelloWorld.java (the simplest Java file, and specified in the sample task file)

You must place all these files in the same directory. Use the following command script to compile and link the sample launcher:

```
#!/bin/sh
#
# To modify this file for your own use, replace all occurrences
# of /usr/lpp/java/IBM with the path to your JDK installation, if different\
#
c89 -c -D USE_390 -W "0,langlvl(extended)" -W c,ss -W c,dll -W "c,float(ieee)"
-I /usr/lpp/java/IBM/J1.3/include -o LauncherFuncs.o LauncherFuncs.c
c89 -c -D USE_390 -W "0,langlvl(extended)" -W c,ss -W c,dll -W "c,float(ieee)"
-I /usr/lpp/java/IBM/J1.3/include -o go.o go.c
c89 -c -D USE_390 -W "0,langlvl(extended)" -W c,ss -W c,dll -W "c,float(ieee)"
-I /usr/lpp/java/IBM/J1.3/include -o jvmcreate.o jvmcreate.c
```

Sample launcher for a JVMSet

```
c89 -o go go.o LauncherFuncs.o
c89 -o jvmcreate jvmcreate.o LauncherFuncs.o
/usr/lpp/java/IBM/J1.3/bin/classic/libjvm.x
```

Execute the sample launcher as follows:

```
go <task file name> <log file for go.c to use>
```

Each JVM creates a log file called jvm.N.log for return codes and error messages.

How the launcher works

This JVMSet launcher uses the spawn function to create child processes that in turn create JVMs and request tasks to be executed. The program is divided into two main pieces:

- go.c
- jvmcreate.c

go.c is the launcher and monitor of the JVMSet. go.c does the initial work of reading and parsing the properties file to get the options to create each JVM. It also initializes and creates the following:

- Semaphore set
- Shared memory segment
- Task list
- JVM creation processes

The shared memory segment is used to pass certain information easily between go.c and the child jvmcreate.c processes. The shared memory segment includes:

- The semaphore set ID needed to use the semaphores created in go.c by the child processes
- The task list
- An array of the reset intervals of all JVMs, indexed by an ID that is assigned on creation of that JVM
- The token

The task list is a linked list consisting of nodes, each containing:

- A class name
- The arguments to be passed to that class upon execution
- The pointer to the next node of the list

The semaphore set ID is an integer that is used to identify the set of semaphores to be used. All child processes use this same set ID so that everyone is using the same semaphores. After these structures are created, and initialized with the needed data, go.c begins spawning jvmcreate.c child processes. These processes are where the JVMs are created and where they request and do work.

go.c invokes jvmcreate.c by a call to spawn(). Jvmcreate.c then sets itself up to access the shared memory segment. After parsing the arguments that were sent to it, jvmcreate.c decides what to do next by checking its ID. ID 0 is the master JVM, which calls `JNI_CreateJavaVM`, and then sets the token returned by this call in the shared memory segment. The master then notifies go.c that its creation is complete using one of the semaphores in the set (denoted by JVM_CREATE_WAIT). The master then goes to sleep on the MASTER_WAIT semaphore, awaiting a signal from go.c to shutdown. If the ID is nonzero, the JVM is considered a worker. A worker JVM will first get the token the master has set from the shared memory, and place it in its JavaVMInitArgs array. After the token is set in this array, `JNI_CreateJavaVM` is called. After completion, it notifies go.c that it has finished creating the JVM, and calls `Execute()`.

The Execute function is where the worker obtains tasks to be run, and runs them. It first checks to make sure there are tasks in the list by way of the WAIT_FOR_TASK semaphore. If the check is successful, the process continues and locks the task list (TASK_LIST_MUTEX semaphore) to gain exclusive access to the task list so it can modify it. The first task in the list becomes the task to be executed, and the next task is moved to the front of the list. It then checks the task to see if it is a "shutdown" task, which will cause this JVM to shut down. If it is not, it locates the class (`FindClass()`), gets the method ID of the main function of this class (GetStaticMethodID), and calls the main function (`CallStaticVoidMethod()`). If the reset interval is met, the JVM resets after returning from calling the main function. All of this repeats infinitely until the Execute function catches one of the special "shutdown" tasks that are placed at the end of the task list by go.c after adding all of the specified classes to the list.

While the JVM threads are executing tasks, go.c simply waits for each child process to end and catches its exit code for logging. After all workers have exited, go.c tells the master to shut down (the master cannot catch a shutdown task because it does not request tasks from the task list).

Notes:

1. The **-Xjvmset** option must be specified for both master and worker JVMs. The master JVM can include the size after the option, for example, **-Xjvmset10M**, to specify the total size of the shared memory segment. The worker JVM specifies the option as **-Xjvmset**.
2. Token handling:

   When calling `JNI_CreateJavaVM`, one of the arguments is a JavaVMInitArgs structure. In this structure is an array of JavaVMOption, called options. Each JavaVMOption structure contains the fields optionString and extraInfo. OptionString is where the string of the particular option (-Dsomeoption or -Xsomeoption, and so on) is placed. The extraInfo field is where you get and set the token. The token is get and set from the extraInfo field coinciding with the **-Xjvmset** optionString in the options array.

   For a master, **-Xjvmset10M**, for example, is placed in options[n].optionString, and options[n].extraInfo is left NULL before calling `JNI_CreateJavaVM`. The extraInfo field is filled with the token on successful completion of the JVM creation.

   For a worker, **-Xjvmset** option is placed in options[n].optionString, but must also have the token that was created by the master placed in options[n].extraInfo before creation.

go.c

```
#include "LauncherHeader.h"
#include <spawn.h>
#include <sys/wait.h>
#include <sys/modes.h>

FILE *fp;
int main(int argc, char *argv[]) {

char *shmIdStr;                       //String representation of shared memory ID
union semun sem_val;
```

```
int i,x,q;                                        //Loop counters
struct inheritance inherit;
const char *c_argv[MAX_ARGS], *c_envp[15];  //For passing Options to spawn()ed process
char buf[256];
char *shmAddr;                                    //Address of shared memory (beginning)
int shmId;                                        //shared memory segment ID
struct shmid_ds shmDesc;               //required for de-allocation of shared memory
pid_t pid,child;              //Used for holding pids returned by spawn() and wait()
time_t t;
int status;                                       //Exit status
int numJVMS;                                      //Number of total JVMs, master+workers
jvmOptionsStruct *JVMS;                           //Holds options for all JVMs to be created
char *SHUTDOWN;                                   //String, defined as SHUTDOWN_TASK
char IDstr[16];                  //String to pass ID to jvmcreate processes via c_argv
char *goLogFname;
char *classesFileName;

SHUTDOWN = strdup("SHUTDOWN_TASK");
classesFileName = argv[1];
goLogFname = argv[2];
if((classesFileName == NULL) || (goLogFname == NULL)) {
        printf("Usage: go <name of file with classes and arguments> <name of log file
to log progress to> \n");
        exit(1);
}

fp = fopen(goLogFname,"w");

//****************************************************
//*****First get shared memory segment created!*****
//****************************************************

shmId = shmget(IPC_PRIVATE,sizeof(sharedDataStruct),S_IRUSR | S_IWUSR); //Allocate
                                              // shared memory for this program
//But allow read/write access for those who attach
if(shmId == -1) {
perror("shmget: ");
exit(1);
}
fprintf(fp,"Shared memory segment allocated.\n");
//Attach Shared memory segment to this process
shmAddr = shmat(shmId,NULL,0);
if(!shmAddr) {
perror("shmatt: ");
exit(1);
}
fprintf(fp,"Shared memory segment attached to this process.\n");


//**************************
//**Now make semaphore set!**
//**************************

sem_set_id = semget(IPC_PRIVATE, 4,S_IWUSR | S_IRUSR); //Create a set of 4 semaphores,
                                                     //allowing others to modify them
if (sem_set_id == -1) {
perror("main: semget");
exit(1);
    }
fprintf(fp,"Semaphore set created, semaphore set id '%d' \n",sem_set_id);
//Initialize Semaphore set:
sem_val.val = 0;
semctl(sem_set_id,WAIT_FOR_TASK,SETVAL,sem_val);
semctl(sem_set_id,MASTER_WAIT,SETVAL,sem_val);
semctl(sem_set_id,JVM_CREATE_WAIT,SETVAL,sem_val);
sem_val.val = 1;
semctl(sem_set_id,TASK_LIST_MUTEX,SETVAL,sem_val);
```

```
//Get ready to use shared memory.
data = (sharedDataStruct *)shmAddr;   //Create a way to access the data in shared memory.
data->sem_set_id = sem_set_id;
data->taskList = &taskList;

//Get settings for go.c
getMonitorSettings("go.prp");

numJVMS = WorkersCount+1;

//Allocate memory for the array of all jvmOptionsStructs
JVMS = memAlloc(numJVMS*sizeof(jvmOptionsStruct));

//Allocate memory for the shared array of resetIntervals (to point to)
data->resetIntervals = memAlloc(numJVMS*sizeof(int));

//Initialize all the options structs:
  for(i = 0; i < numJVMS; i++) {
if(!i)
initOptionsStruct(&JVMS[i],"master",0);
else
  initOptionsStruct(&JVMS[i],"worker",i);

  getProperties(&JVMS[i],"go.prp");
  parseJvmArgs(&JVMS[i],JVMS[i].DOpts);
  parseJvmArgs(&JVMS[i],JVMS[i].XOpts);

  }

fprintf(fp,"All JVM options read and parsed.\n");
//Set all the resetIntervals:
for(i=0;i<numJVMS;i++)
        data->resetIntervals[i] = JVMS[i].resetInterval;

//Initialize the semaphores to proper values:
sem_val.val = 0;
semctl(sem_set_id,WAIT_FOR_TASK,SETVAL,sem_val);
sem_val.val = 1;
semctl(sem_set_id,TASK_LIST_MUTEX,SETVAL,sem_val);

//Time to create all the JVMS!
memset(buf,0,255);
sprintf(buf,"%d",shmId);
shmIdStr = strdup(buf);                //Make a string version of the shmID to pass on
x = 0;

for(i=0;i<envVarCount;i++)
        fprintf(fp,"Environment Variable #%d: %s \n",i,envVars[i];

//Build argument list for child processes
for(i=0;i<numJVMS;i++) {
        c_argv[x++] = LAUNCHER;
     fprintf(fp,"\nOptions for JVM #%d: \n",i);
     for(q = 0; q< JVMS[i].numJvmOpts;q++){
         fprintf(fp,"Option[%d]=%s \n",q,JVMS[i].jvmArgs[q]);
         c_argv[x++] = JVMS[i].jvmArgs[q];
}
     c_argv[x++] = shmIdStr; //tack on shared memory ID
     memset(IDstr,0,16);
     sprintf(IDstr,"%d",i);
     c_argv[x++] = IDstr;    //tack on an ID for the JVM
     c_argv[x] = NULL;

/* Build the environment structure which defines the child's environment variables */
for(x=0;x<envVarCount;x++) {
           c_envp[x] = envVars[x];
```

```
    }

    c_envp[x] = NULL;

  //spawn the child thread:
  child=spawn(LAUNCHER, 0, NULL, &inherit, c_argv, c_envp);
    if(child==-1) {
  perror("Error on spawn");
  exit(-1); }
    else
  printf("Spawned %i\n", child);

  printf("JVM #%d has been spawned! \n",i);
  fprintf(fp,"JVM[%d] has a pid of: %d \n",i,child);
  waitForJVM();
  memset(c_argv,0,MAX_ARGS);
  x = 0;
  }
  //All Jvms created and waiting for tasks to execute, time to create the task list:
  getClasses(classesFileName);
  for(i=0;i<WorkersCount;i++) {
          lockTaskList();
                  addTask(SHUTDOWN,NULL);
                  taskAvailable();
          releaseTaskList();
  }

  //Add a special termination "task" for each worker to catch to the end of the list.

  printf("Done populating task list.... \n");

  //Job is done, sit back and wait for each worker to end, catching exit statuses
  for(i=0;i<numJVMS;i++) {
          if(i == WorkersCount)
                  killMaster();  //All workers have exited, now kill the master.
  if((pid = wait(&status)) == -1)
  perror("wait() error");
  else {
  time(&t);
  printf("Launcher has detected a JVM shutting down. \n");
  printf("The pid of the proccess that ended was %d.\n",(int)pid);
  if(WIFEXITED(status))
  fprintf(fp,"Proccess %d ended normally with status of %d \n",pid,WEXITSTATUS(status));
  else if(WIFSIGNALED(status))
  fprintf(fp,"Process %d was TERMINATED by signal %d \n",pid,WTERMSIG(status));
  else if(WIFSTOPPED(status))
  fprintf(fp,"Proccess %d was STOPPED by signal %d\n",pid,WSTOPSIG(status));
  else
  fprintf(fp,"**Process %d terminated for an unknown reason!! \n",pid);
  }

  }

  /***********Free All dynamically allocated memory *********************************/
  for(i = 0; i<numJVMS;i++) {
          free(JVMS[i].prefix);
       free(JVMS[i].jvmArgs);
       free(JVMS[i].XOpts);
       free(JVMS[i].DOpts);
  }

  for(i=0;i<envVarCount;i++)
          free(envVars[i]);

  clearList(taskList);

  //Shared memory cleanup:
```

```
free(shmIdStr);
if(shmdt(shmAddr) == -1)
perror("shmdt: ");
if(shmctl(shmId,IPC_RMID, &shmDesc) == -1)
perror("main: shmctl: ");

//Misc. cleanup:

free(envVars);
free(JVMS);
free(LogName);

return 0;
}
```

LauncherHeader.h

```
  /******************************/
 /****** Defines **************/
/******************************/
#define JVMSET
//This needs to be defined for using shared classes/handling the token
#define _XOPEN_SOURCE_EXTENDED 1
#define _POSIX_SOURCE
//Required to use some functions in the includes
#define NEXT_OPTION '-'
#define NEW_LINE '\n'
#define MAX_CHARS_PER_LINE 500
#define MAX_ARG_LENGTH 250
#define WHITE_SPACE ' '
#define EQUAL_STRINGS 0
#define TRUE 1
#define FALSE 0
#define PROPERTIES_COMMENT_MARKER '#'
#define MAX_ARGS 25
//The following are IDs for each semaphore that is created in go.c
#define TASK_LIST_MUTEX 0
#define WAIT_FOR_TASK 1
#define MASTER_WAIT 2
#define JVM_CREATE_WAIT 3

#define LAUNCHER "jvmcreate"
#define NULL_CHAR '\0'

//Master is created as 0 ID, using this for readability
#define MASTER 0

#define TRUE 1
#define FALSE 0
#define JVM_MAX_OPTS 20
  /***************************/
 /***** Common Includes ******/
/***************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <sys/sem.h>
  /**************************/
 /**** Structs/Unions  ******/
/**************************/
//This is a single structure to easily manage various options associated with a
// particular JVM
typedef struct {

        char *prefix; //set this to "master" or "worker"
```

```
        int Id;                     //set this to 0 for master, or n for worker.n
        char *DOpts,*XOpts;
        char **jvmArgs;             //Array of individual options for the jvm, used to
                                    //easily create arguments array for a spawned JVM.
        int numJvmOpts,maxJvmOpts;  //Used in the parsing of DOpts,XOpts into the
                                    //jvmArgs array
int resetInterval;          //Defines how often to reset (Every resetInterval tasks)

} jvmOptionsStruct;

union semun {                       //Used for manipulating semaphores at creation
          int              val;
          struct semid_ds *buf;
          unsigned short  *array;
} arg;

typedef struct {                    //An individual node containing information needed
                                    //to execute a task and pointing to the next task
        char *className;
        char *classArgs;
        void *nextTask;
} taskNode;

typedef struct {                   //The structure that is used to define the format of
                                   //the shared memory segment
        taskNode **taskList;       //The universal list of tasks to be done
        int sem_set_id;            //The id of the semaphore set
        void *token;               //Used to get the token from master to workers
        int *resetIntervals;       //An array containing all of the resetIntervals,
                                   //indexed by ID (0 for master, n for worker.n)
} sharedDataStruct;

  /******************************/
 /**** Globals *****************/
/******************************/

int sem_set_id;  //Global for semaphore set ID
taskNode *taskList;  //Global holding the list of classes to execute
int WorkersCount;       //Global for the number of Workers  (Used only by go.c)
char **envVars;         //Global for the environment variables to be set in all spawned
                        //JVMs (go.c)
int envVarCount;        //Global for how many environment variables there are (go.c)
char *LogName;
sharedDataStruct *data;  //Global for referencing shared memory

  /****************************/
 /**** Functions *************/
/****************************/

//Get Properties:
// Send in a jvmOptionsStruct that will have its fields filled in by the values in
//properties file "filename"
//  Note: This struct is initialized by initOptionsStruct() first.
void getProperties(jvmOptionsStruct *jvmOpts, char *filename);

//parseJvmArgs:
// Takes a filled in jvmOptionsStruct* (from getProperties()), and adds each option
// of a string of arguments to
// seperate elements of an array. Used on DOpts and XOpts to add to the jvmArgs array
void parseJvmArgs(jvmOptionsStruct *jvmOpts,char *args);

//addOption:
// Helper function for parseJvmArgs. Manages the size of the jvmArgs array, expands
// if necessary, and puts the option into the array
// char ***Opts is (&jvmArgs) from a jvmOptionsStruct, *str is what is to be
// added to the array
// *numOpts and *maxOpts are (&numJvmOpts) and (&maxJvmOpts) from a
```

```
// jvmOptionsStruct, respectively
int addOption(char ***Opts, char *str, int *numOpts, int *maxOpts);

//getMonitorSettings:
// Takes the Properties file "filename" and extracts options for go.c to use.
// Options include: Workers count, Environment variables (and number of), and Log name
void getMonitorSettings(char *filename);

//getClasses:
// Reads "filename" and extracts class names and arguments for creating a list of tasks
// to be run by the worker Jvms.
void getClasses(char *filename);

//initOptionsStruct:
// takes a jvmOptionsStruct, a "prefix" (worker/master),an id, and initializes
// all values.
void initOptionsStruct(jvmOptionsStruct *jvmOpts, char *prefix, int id);
//memAlloc:
// Helper function, ensures memory is allocated and gives an error
// if something went wrong
// Used just like malloc, but error checked.
void *memAlloc(size_t size);

//The following functions are used to more easily handle the various semaphores.
void waitForTask();    //Wait for a task to enter the task list, sleep until then.
void taskAvailable();  //Used by the main program to wakeup and allow passage to
                       //threads seeking tasks
void lockTaskList();   //Used for Mutex on the shared list of tasks.
void releaseTaskList();//Used to release the mutex.
void killMaster();     //Used to tell the Master JVM thread to quit.
void waitForTask();    //Used to check if a task is waiting to be run, and makes the
                       //thread wait if no taks are waiting
void masterWait();     //Used to put the master to sleep until killMaster() is called
void jvmAvailable();   //Used to notify go.c that JNI_CreateJavaVM() has finished
                       //in jvmcreate
void waitForJVM();     //Used by go.c to wait for the creation of a JVM in a child
                       //process to finish

//The following are used to manipulate the task list
void advanceList(taskNode **list);         //Moves the front of the list to the next task
//Caller is responsible for freeing the previously leading Node

void addTask(char *className, char *classArgs); //Used to add a new taskNode to the list
                                    //with className and classArgs set to supplied values.
void clearList();  //Used at the end of go.c to clean up whatever may be left in the list
```

LauncherFuncs.c

```
#include "LauncherHeader.h"

  /*********************************************/
 /******** Funcs for Semaphore operations *****/
/*********************************************/
void waitForTask() {
        struct sembuf semOp;
        semOp.sem_num = WAIT_FOR_TASK;
        semOp.sem_op = -1;
        semOp.sem_flg = 0;
        semop(sem_set_id, &semOp,1);
}

void taskAvailable() {
        struct sembuf semOp;
        semOp.sem_num = WAIT_FOR_TASK;
        semOp.sem_op = 1;
```

```
        semOp.sem_flg = 0;
        semop(sem_set_id, &semOp,1);
}


void lockTaskList() {
        struct sembuf semOp;
        semOp.sem_num = TASK_LIST_MUTEX;
        semOp.sem_op = -1;
        semOp.sem_flg = 0;
        semop(sem_set_id, &semOp,1);


}


void releaseTaskList() {
        struct sembuf semOp;
        semOp.sem_num = TASK_LIST_MUTEX;
        semOp.sem_op = 1;
        semOp.sem_flg = 0;
        semop(sem_set_id,&semOp,1);
}

void killMaster() {
        struct sembuf semOp;
        semOp.sem_num = MASTER_WAIT;
        semOp.sem_op = 1;
        semOp.sem_flg = 0;
        semop(sem_set_id, &semOp,1);


}
void waitForJVM() {
        struct sembuf semOp;
        semOp.sem_num = JVM_CREATE_WAIT;
        semOp.sem_op = -1;
        semOp.sem_flg = 0;
        semop(sem_set_id, &semOp,1);


}

void masterWait() {
        struct sembuf semOp;
        semOp.sem_num = MASTER_WAIT;
        semOp.sem_op = -1;
        semOp.sem_flg = 0;
        semop(sem_set_id, &semOp,1);


}
void jvmAvailable() {
        struct sembuf semOp;
        semOp.sem_num = JVM_CREATE_WAIT;
        semOp.sem_op = 1;
        semOp.sem_flg = 0;
        semop(sem_set_id, &semOp,1);


}

  /**************************************************/
 /****** Task List Manipulating Functions *********/
/************************************************/
void addTask(char *className, char *classArgs) {
        taskNode *iter,*temp;
```

```
        taskNode *newNode;
        newNode = memAlloc(sizeof(taskNode));
        newNode->nextTask = NULL;
        newNode->className = strdup(className);
        if(classArgs == NULL)
                newNode->classArgs = NULL;
        else
                newNode->classArgs = strdup(classArgs);
#ifdef USE_390
//The special "task" of SHUTDOWN_TASK, must not be converted to ascii
        if(strcmp(className,"SHUTDOWN_TASK") != EQUAL_STRINGS) {
                __etoa(newNode->className);
                if(classArgs != NULL)
                        __etoa(newNode->classArgs);
        }
#endif
        iter = taskList;
        if(iter == NULL){
                 taskList = newNode;
        }
        else {
                while((iter->nextTask) != NULL) {
                        iter = iter->nextTask;
                }
                iter->nextTask = newNode;
        }
}

void clearList() {
        taskNode *iter;
        taskNode *temp;

        if(taskList == NULL)
                return;
        iter = taskList;
        while(iter != NULL) {
                temp = iter->nextTask;
                free(iter->className);
                free(iter->classArgs);
                free(iter);
                iter = temp;
        }
        return;
}


void advanceList(taskNode **list) {
        (*list) = (*list)->nextTask;
}

void initOptionsStruct(jvmOptionsStruct *jvmOpts, char *prefix,int id) {
        jvmOpts->prefix = strdup(prefix);
        jvmOpts->Id = id;
        jvmOpts->numJvmOpts = 0;
        jvmOpts->maxJvmOpts = 0;
        jvmOpts->jvmArgs = NULL;
        return;

}

  /******************************/
 /***** Parsing Functions *******/
/******************************/

void parseJvmArgs(jvmOptionsStruct *jvmOpts,char *args) {

        int i,j,x;
```

```
                char tmp[MAX_ARG_LENGTH];


                i = j = x = 0;

                while(args != NULL && args[i] != NULL_CHAR) { //Until end of string
                        if(args[i] == NEXT_OPTION) {  //Next options is signified by
                                                      // a '-' character
                                do {
                                        tmp[j++] = args[i++];
                                   }
                                while(args[i] != NEXT_OPTION && args[i] != NULL_CHAR
                               && args[i] != WHITE_SPACE);

                        while(args[i] == WHITE_SPACE)
                                i++;

                        tmp[j] = NULL_CHAR;

                        addOption(&(jvmOpts->jvmArgs),tmp,&(jvmOpts->numJvmOpts),
                        &(jvmOpts->maxJvmOpts)); //Add extracted option to jvmArgs
                        memset(tmp,0,MAX_ARG_LENGTH);             //reinitialize tmp

                        j = 0;

                        } //End if

                        else
                                i++;
                } //End While

                return;
        }       //End parseJvmArgs

        void getClasses(char *filename)
        {
                FILE *fp;
                char tmp[50];
                char buffer[MAX_CHARS_PER_LINE];
                unsigned short flag = 0;
                char className[60];
                char classArgs[440];
                int i, j,x,n,argLen;   //i = buffer index, j = field index, n = id index,
                                       // argLen = value index, x = loopcounter

                i = j = x = argLen = n = 0;

                fp = fopen(filename,"r");

                if(fp == NULL) {
                        printf("Failed to open %s. \nEnsure Properties file is in the
                               correct location.\n",filename);
                        return;
                }

        lockTaskList();                  //To ensure no-one tries to manipulate the
                                         //list while it is being modified
                while(fgets(buffer,MAX_CHARS_PER_LINE,fp) != NULL) { //Grab one line
                                                                     //at a time from the file

                        i=0;
                        x=0;
                        while(buffer[i] != WHITE_SPACE && buffer[i] != NEW_LINE) {
                                className[i] = buffer[i];
                                i++;
                        }
                        className[i] = NULL_CHAR;
```

```
                if(buffer[i] == WHITE_SPACE) {
                        i++;
                        while(buffer[i] != NEW_LINE)
                                classArgs[x++] = buffer[i++];
                        classArgs[x] = NULL_CHAR;
                }
                else
                        classArgs[0] = NULL_CHAR;

                addTask(className,classArgs); //Now add to the task list
                taskAvailable();              //Notify that there is a task to be run

//Reinitialize for another iteration:
memset(buffer,0,MAX_CHARS_PER_LINE);
                memset(className,0,60);
                memset(classArgs,0,440);
        }
releaseTaskList();
//Done modifying task List, release lock and return
return;
}

void getMonitorSettings(char *filename) {

        FILE *fp;
        char buffer[MAX_CHARS_PER_LINE];
        char p1[MAX_ARG_LENGTH],p2[MAX_ARG_LENGTH],p3[MAX_ARG_LENGTH];
        char value[MAX_ARG_LENGTH];
        int i,j,k;

i = j = k = 0;
fp = fopen(filename,"r");
while(fgets(buffer,MAX_CHARS_PER_LINE,fp) != NULL) { //Grab a full line

  if(buffer[0] != PROPERTIES_COMMENT_MARKER) {         //Lines headed with a
                                                       //PROPERTIES_COMMENT_MARKER
                                                       //are ignored

while(buffer[i] != NULL_CHAR && buffer[i] != NEW_LINE && buffer[i]
    != '=' && buffer[i] != '.') {
                p1[i] = buffer[i];
                i++;
        }
        p1[i] = NULL_CHAR;
        if(buffer[i] == '.') {
                i++;
                while(buffer[i] != NULL_CHAR && buffer[i] != NEW_LINE &&
                 buffer[i] != '=' && buffer[i] != '.')
                        p2[j++] = buffer[i++];

                p2[j] = NULL_CHAR;
                j = 0;
                }

        if(buffer[i] == '.') {
                i++;
                while(buffer[i] != NULL_CHAR && buffer[i] != NEW_LINE &&
                 buffer[i] != '=')
                        p3[j++] = buffer[i++];
                p3[j] = NULL_CHAR;
                j=0;
        }
        if(buffer[i] == '=') {
                i++;
                while(buffer[i] != NULL_CHAR && buffer[i] != NEW_LINE)
                        value[k++] = buffer[i++];
                value[k] = NULL_CHAR;
```

```
        }
        if((strcmp(p1,"env") == EQUAL_STRINGS) && (strcmp(p2,"var") == EQUAL_STRINGS)){
                if(strcmp(p3,"count") == EQUAL_STRINGS) {
                        envVarCount = atoi(value);
                        envVars = memAlloc(envVarCount * sizeof(char *));
                }
                else
                if((atoi(p3) <= envVarCount) && (envVarCount))
                        envVars[atoi(p3)-1] = strdup(value);
                else
                        printf("Error in file format, please specify env.var.count
                                before listing variables.\n");
        }
        else
        if(strcmp(p1,"log") == EQUAL_STRINGS){
                if(strcmp(p2,"name") == EQUAL_STRINGS)
                        LogName = strdup(value);

        }
        else
        if(strcmp(p1,"workers") == EQUAL_STRINGS) {
                if(strcmp(p2,"count") == EQUAL_STRINGS)
                        WorkersCount = atoi(value);
        }

buffer[i] = NULL_CHAR;
//Re-initialize
memset(buffer,0,MAX_CHARS_PER_LINE);
memset(p1,0,MAX_ARG_LENGTH);
memset(p2,0,MAX_ARG_LENGTH);
memset(p3,0,MAX_ARG_LENGTH);
memset(value,0,MAX_ARG_LENGTH);

i = j = k = 0;

  }//end Properties comment marker if
}//end while loop
fclose(fp);
}//end function

void getProperties(jvmOptionsStruct *jvmOpts, char *filename)
{
        FILE *fp;
        char tmp[50];
        char buffer[MAX_CHARS_PER_LINE];
        char field[MAX_ARG_LENGTH];
        char prefix[10];
        char Id[6];
        char value[MAX_ARG_LENGTH];
        unsigned short flag = 0;

        int i, j,x,n,argLen;   //i = buffer index, j = field index, n = id index,
                               //argLen = value index, x = loopcounter

        i = j = x = argLen = n = 0;

        fp = fopen(filename,"r");

        if(fp == NULL) {
                printf("Failed to open %s. \nEnsure Properties file is in the correct
                       location.\n",filename);
                return;
        }

while(fgets(buffer,MAX_CHARS_PER_LINE,fp) != NULL) {

  if(buffer[0] != PROPERTIES_COMMENT_MARKER) {
```

```
        while(buffer[i] != NULL_CHAR  && buffer[i] != '.' && buffer[i] != '=') {
               prefix[i] = buffer[i];
                 i++;
        }
        prefix[i++] = NULL_CHAR; // Skip "."


        if((strcmp(prefix,"worker") == EQUAL_STRINGS) &&
          (strcmp(prefix,jvmOpts->prefix) == EQUAL_STRINGS))
        {
                while(buffer[i] != NULL_CHAR && buffer[i] != '.')
                        Id[n++] = buffer[i++];

                Id[n] = NULL_CHAR;
                i++; // Skip "."
                if(atoi(Id) == jvmOpts->Id) //Are we talking about the same worker?
                        flag = TRUE;
                else
                        flag = FALSE;
        }
        else
        {
                if(strcmp(prefix,jvmOpts->prefix) == EQUAL_STRINGS) //not a worker,
                                                                    //is it master?
                        flag = TRUE;
                else
                        flag = FALSE;
        } // end else

        if(flag) {
                while(buffer[i] != '=')
                        field[j++] = buffer[i++];
                field[j] = NULL_CHAR;
                i++; //Skip '='


                while(buffer[i] != NEW_LINE && buffer[i] != NULL_CHAR)

                       value[argLen++] = buffer[i++];

                value[argLen] = NULL_CHAR;
                buffer[i] = NULL_CHAR;


                if(strcmp(value," ") == EQUAL_STRINGS)
                        value[0] = NULL_CHAR;
else
                  if(strcmp(field,"options") == EQUAL_STRINGS)
                     jvmOpts->DOpts = strdup(value);
                else
                  if(strcmp(field,"options.properties") == EQUAL_STRINGS)
                        jvmOpts->XOpts = strdup(value);
else
  if(strcmp(field,"reset.interval") == EQUAL_STRINGS)
jvmOpts->resetInterval = atoi(value);

        } //end if(flag)

        memset(buffer,0,MAX_CHARS_PER_LINE);
        memset(field,0,MAX_ARG_LENGTH);
        memset(value,0,MAX_ARG_LENGTH);
        memset(tmp,0,50);
        memset(Id,0,6);
        memset(prefix,0,10);

        i=0;
```

```
        j=0;
        n = 0;
        argLen = 0;
        flag = 0;
  }//end if(buffer[0] != '#')

} //end while(fgets)

fclose(fp);
return;
}

int addOption(char ***Opts, char *str,int *numOpts,int *maxOpts)
{
    /*
     * Expand options array if needed to accomodate at least one more
     * class option.
     */
   char **tmp;
   char temp[30];

        if ((*numOpts) >= (*maxOpts)) {
        if (*Opts == NULL) {
             (*maxOpts) = 4;
            (*Opts) = memAlloc((*maxOpts) * sizeof(char *));
        } else {
            (*maxOpts) *= 2;
            tmp = memAlloc((*maxOpts) * sizeof(char *));
            memcpy(tmp, *Opts, (*numOpts) * sizeof(char *));
            free((*Opts));
            (*Opts) = tmp;
        }
    }
        (*Opts)[(*numOpts)] = strdup(str);

    (*numOpts)++;
    return *numOpts - 1;
}

/***************************************************************************
 * name         - MemAlloc
 * description  - Returns a pointer to a block of at least 'size' bytes
 *                of memory. Prints error message and exits if the memory
 *                could not be allocated.
 * parameters   - size  Size of memory requested
 * returns      - Pointer to block of allocated memory.
 ***************************************************************************/
void *memAlloc(size_t size)
{
        void *p = malloc(size);
        if (p == 0) {
        perror("malloc");
        exit(1);
    }
    return p;
}
```

jvmcreate.c

```
#include "LauncherHeader.h"
#include <jni.h>
#include <errno.h>

#ifdef USE_390
#include <unistd.h>
#endif
```

```
int AddOption(char *str, void *info);
jboolean ParseArguments(int *pargc, char ***pargv,int *pret);
void SetClassPath(char *s);
void Execute();
void *token = NULL;
JavaVMOption *options;
int numOptions = 0;
int maxOptions = 0;
int ShmId;
FILE *fp;
JavaVM *jvm;
int Interval;
char *mainStr    = "main";
char *sigVoid        = "()V";
char *sigString = "([Ljava/lang/String;)V";
int ID;
int jvmSetOption = -1;
JNIEnv *env;
char fname[20];
int main( int argc, char *argv[] ) {

JavaVMInitArgs jvm_args;
jint rc;
int ret;
void *shmAddr;

argv++; //Skip progname
argc--;


if(!ParseArguments(&argc,&argv,&ret)) //Parse argv[]
return ret;

shmAddr = shmat(ShmId,NULL,0);        //Attach shared memory to this process
sprintf(fname,"jvm.%d.log",ID);
fp = fopen(fname,"w");
if((int)shmAddr == -1) {               //Catch attaching error
switch(errno) {
case EACCES: {
printf("EACCES!\n");
break;
}
case EINVAL: {
printf("EINVAL!\n");
break;
}
case EMFILE: {
printf("EMFILE!\n");
break;
}
case ENOMEM: {
printf("ENOMEM!\n");
break;
}
}
exit(1);
}

data = (sharedDataStruct *) shmAddr;   //Define a way to access values in shared memory
sem_set_id = data->sem_set_id;         //Get the semaphore set ID
Interval = data->resetIntervals[ID];   //Get the reset interval for this jvm
token = data->token;
#if defined(USE_390)
__etoa(mainStr );
__etoa(sigVoid );
__etoa(sigString );
```

```
#endif

jvm_args.version  = 0x00010002;  //Build the structure to create the jvm with
jvm_args.options = options;
jvm_args.nOptions = numOptions;

if(ID != MASTER){
                jvm_args.options[jvmSetOption].extraInfo = (data->token);
        }

rc = JNI_CreateJavaVM(&jvm, (void **) &env, &jvm_args );
fprintf(fp,"JVM[%d]: JNI_CreateJavaVM returned %d \n",ID,rc);
if(rc) {
fprintf(fp,"JVM[%d]: JNI_CreateJavaVM failed! Shutting down. \n",ID);
exit(1);
}
if(ID == MASTER) {
                data->token = (options[jvmSetOption].extraInfo);
}
jvmAvailable();

if(ID == MASTER)
masterWait();
else
Execute();




rc = (*jvm)->DestroyJavaVM(jvm);
fprintf(fp,"JVM[%d]: DestroyJavaVM returned rc = %i\n", ID,rc);

return 0;

}

/***************************************************************************
 * name         - ParseArguments
 * description  - Parses command line arguments.
 * parameters   - pargc
 *                pargv
 *                pjarfile
 *                pclassname
 *                pret
 * returns      - JNI_FALSE or JNI_TRUE
 ***************************************************************************/
jboolean ParseArguments(int *pargc, char ***pargv,int *pret)
{
    int argc = *pargc;
    char **argv = *pargv;
    jboolean jarflag = JNI_FALSE;
    char *arg;
    *pret = 1;
    while ((arg = *argv) != 0 && *arg == '-') {
        argv++; --argc;
       if (strcmp(arg, "-classpath") == 0 || strcmp(arg, "-cp") == 0) {
            if (argc < 1) {
                fprintf(stderr, "%s requires class path specification\n", arg);
                return JNI_FALSE;
            }
            SetClassPath(*argv);
            argv++; --argc;
} else
#ifndef OLDJAVA

if (strcmp(arg, "-jar") == 0)
             jarflag = JNI_TRUE;
```

```
else
#endif
/*
 * The following case provide backward compatibility with old-style
 * command line options.
 */
        if (strcmp(arg, "-verbosegc") == 0) {
            AddOption("-verbose:gc", NULL);
        } else if (strcmp(arg, "-t") == 0) {
            AddOption("-Xt", NULL);
        } else if (strcmp(arg, "-tm") == 0) {
            AddOption("-Xtm", NULL);
        } else if (strcmp(arg, "-debug") == 0) {
            AddOption("-Xdebug", NULL);
        } else if (strcmp(arg, "-noclassgc") == 0) {
            AddOption("-Xnoclassgc", NULL);
        } else if (strcmp(arg, "-Xfuture") == 0) {
            AddOption("-Xverify:all", NULL);
        } else if (strcmp(arg, "-verify") == 0) {
            AddOption("-Xverify:all", NULL);
        } else if (strcmp(arg, "-verifyremote") == 0) {
            AddOption("-Xverify:remote", NULL);
        } else if (strcmp(arg, "-noverify") == 0) {
            AddOption("-Xverify:none", NULL);
        } else if (strncmp(arg, "-prof", 5) == 0) {
            char *p = arg + 5;
            char *tmp = memAlloc(strlen(arg) + 50);
            if (*p) {
                sprintf(tmp, "-Xrunhprof:cpu=old,file=%s", p + 1);
            } else {
                sprintf(tmp, "-Xrunhprof:cpu=old,file=java.prof");
            }
            AddOption(tmp, NULL);
        } else if (strncmp(arg, "-ss", 3) == 0 ||
                   strncmp(arg, "-oss", 4) == 0 ||
                   strncmp(arg, "-ms", 3) == 0 ||
                   strncmp(arg, "-mx", 3) == 0) {
            char *tmp = memAlloc(strlen(arg) + 6);
            sprintf(tmp, "-X%s", arg + 1); /* skip '-' */
            AddOption(tmp, NULL);
        } else if (strcmp(arg, "-checksource") == 0 ||
                   strcmp(arg, "-cs") == 0 ||
                   strcmp(arg, "-noasyncgc") == 0) {
            /* No longer supported */
            fprintf(stderr,
                    "Warning: %s option is no longer supported.\n",
                    arg);
#ifdef JVMSET
        } else if (strncmp(arg, "-Xresettable",12) == 0) {
            AddOption(arg ,NULL);
    /* Worker JVM */
        } else if (strcmp(arg, "-Xjvmset") == 0) {
                token = (data->token);
            jvmSetOption = AddOption("-Xjvmset",NULL);
        /* Master JVM */
       } else if (strncmp(arg, "-Xjvmset",8) == 0) {
            jvmSetOption = AddOption(arg ,NULL);

} else if (strncmp(arg, "-gc",3) == 0) {
           // gcInterval = atoi(arg+3);
        } else if (strncmp(arg, "-I",2) == 0) {
            //iterations = atoi(arg+2);
        } else if (strncmp(arg, "-R",2) == 0) {
            //resetInterval = atoi(arg+2);
        } else if (strncmp(arg, "-log=",5) == 0) {
            char *logName = arg + 5;
            char *logOption = memAlloc(strlen(arg) + 50);
```

```
                if (*logName) {
                    sprintf(logOption, "-Dibm.jvm.events.output=%s", logName);
                } else {
                    sprintf(logOption, "-Dibm.jvm.events.output=unresettable.log");
                }
        printf("logOption: %s \n",logName);
                AddOption(logOption, NULL);
                AddOption("-Dibm.jvm.unresettable.events.level=max",NULL);
                AddOption("-Dibm.jvm.resettrace.events",NULL);
                AddOption("-Dibm.jvm.crossheap.events",NULL);
#endif
            } else {
                AddOption(arg, NULL);
            }
        }

if (--argc >= 0) {
  ShmId = atoi(*argv);
argv++;
}
if(--argc >= 0) {
ID = atoi(*argv);
argv++;
}

    return JNI_TRUE;
}
/***************************************************************************
 * name         - AddOption
 * description  - Adds a new VM option with the given name and value.
 * parameters   - str   Option string (e.g. "-Djava.class.path=...")
 *                info  Additional information
 * returns      -
 ***************************************************************************/
int AddOption(char *str, void *info)
{
    /*
     * Expand options array if needed to accomodate at least one more
     * VM option.
     */
    if (numOptions >= maxOptions) {
        if (options == 0) {
            maxOptions = 4;
            options = memAlloc(maxOptions * sizeof(JavaVMOption));
        } else {
            JavaVMOption *tmp;
            maxOptions *= 2;
            tmp = memAlloc(maxOptions * sizeof(JavaVMOption));
            memcpy(tmp, options, numOptions * sizeof(JavaVMOption));
            free(options);
            options = tmp;
        }
    }

    #if defined(USE_390)
__etoa_l((void *)str,strlen(str));
   #endif

    options[numOptions].optionString = str;
    options[numOptions++].extraInfo = info;

    return (numOptions-1);
}

/***************************************************************************
 * name         - SetClassPath
 * description  - Set the classpath option using the supplied string
```

```
 * parameters   - s     Null terminated string holding the classpath.
 * returns      -
 *********************************************************************/
void SetClassPath(char *s)
{
    char *def = memAlloc(strlen(s) + 40);
#ifdef OLDJAVA
    sprintf(def, "-Xbootclasspath:%s", s);
#else
    sprintf(def, "-Djava.class.path=%s", s);
#endif

    AddOption(def, NULL);
}

void Execute() {                   //This is where workers request and execute tasks
   jclass myclass;
   jmethodID mid;
   char *className;
   char *classArgs;
   taskNode *deadNode;
   int resetCount = 0;
   int resetCounter = 0;
   int rc = 0;
   while(1) {               //We will break the loop manually when ready
waitForTask();           //See if there are tasks in the list, if not, wait for one
  lockTaskList();        //Here we will get a task and advance the list, locked to
                            //ensure only 1 process modifies the list at a time
className = (*(data->taskList))->className;
classArgs = (*(data->taskList))->classArgs;
deadNode = *(data->taskList);
advanceList(data->taskList);
  releaseTaskList();
if(strcmp(className,"SHUTDOWN_TASK") == 0){   //If the classname is the special
                                                 //SHUTDOWN_TASK, quit
break;
}
myclass = (*env) -> FindClass(env, className);
if (myclass == 0)
{
fprintf(fp, "JVM[%d]: FindClass failed!\n",ID);
break;
    }
mid = (*env) -> GetStaticMethodID(env, myclass,mainStr,sigString);
    if (mid == 0)
    {
fprintf(fp, "JVM[%d]: GetStaticMethodID() failed\n");
break;
    }
(*env) -> CallStaticVoidMethod(env, myclass, mid); //Call the main method of the class
fprintf(fp,"Done executing class... \n");
        resetCounter++;
        if((resetCounter % Interval) == 0) {       //Check if we are supposed to reset
                                                   //this time around
                if ( (*env)->ExceptionOccurred(env) ) {   //Catch exceptions
                                                          //before resetting
                        (*env)->ExceptionDescribe(env);
                        printf("An exception occured!! \n");
                        (*env)->ExceptionClear(env);
                }
                (*env)->DeleteLocalRef(env, myclass);
                rc = (*jvm)->ResetJavaVM(jvm);
                if ( (*env)->ExceptionOccurred(env) ) {
                        (*env)->ExceptionDescribe(env);
                        printf("An exception occured!! \n");
                        (*env)->ExceptionClear(env);
                }
```

```
                        resetCount++;
                        fprintf(fp,"ResetJavaVM returned rc = %i,
                                resetCount = %i\n", rc, resetCount);
                        if (rc != 0) {
                                fprintf(fp,"ResetJavaVM returned non-zero, rc=%d \n",rc);
                                break;
                        }
                }


        free(deadNode);
            }
            fprintf(fp,"JVM[%d]: Detaching and shutting down! \n",ID);
            (*jvm) -> DetachCurrentThread(jvm);

        }
```

go.prp

```
###########################################################
#
#
# To modify this file for your own use:
#
# 1.  Replace all occurrences of /usr/lpp/java/IBM with the path
#     to your JDK installation, if different
# 2.  Replace all occurrences of /u/myclasses with the path to
#     your classes
#
#
# Environment Variables
#
# This section is where we declare environment variables for
# ALL subsequent JVMs that we create.  env.var.count is required
# if any env.var properties are to be used.
#
#
############################################################
env.var.count=2
env.var.1=_BPX_SHAREAS=YES
env.var.2=LIBPATH=/lib:/bin/lib:/usr/lpp/java/IBM/J1.3/bin:
                  /usr/lpp/java/IBM/J1.3/bin/classic
#############################################################
#
#
#
# Monitor
#
# This section is used for miscellaneous properties.  The
# following is a list of all possible properties for this
# section.
#
# log.name={fileName}
# workers.count=
#
##############################################################
log.name=jvmset.log
workers.count=2
##############################################################
#
#
#
#
# Master
#
# This section is used to set various properties for the
```

```
# Master JVM if running a JVMSet.  Following is a list of
# all properties available to set.  -Xjvmset[token] must be
# specified here for this to work!
#
# master.options={-Xargs}
# master.options.properties={-Dargs}
#
#############################################################
master.options=-Xms16M -Xmx32M -Xresettable -Xjvmset10M
master.options.properties=-Dibm.jvm.events.output=stdout.log
                          -Djava.compiler=NONE -Dibm.jvm.unresettable.events
################################################################
#
#
#
#
# Workers
#
# This section is used to set various properties for worker JVMs.
# Worker properties have the worker.n.property=<value> where
# 0 < workers.count <= n.  It is important to have properties set
# up for the 1 -> n worker JVMs.  -Xjvmset must be specified here
# for this to work!  Of course, -Xresettable must be set for
# worker.n.reset.interval=<value> to be meaningful.
#
#
# worker.n.options={-Xargs}
# worker.n.options.properties={-Dargs}
# worker.n.reset.interval=<value>
#
###############################################################
worker.1.options=-Xms16M -Xmx32M -Xresettable -Xjvmset
worker.1.options.properties=-Djava.class.path=/u/myclasses -Djava.compiler=NONE
                            -Dibm.jvm.unresettable.events
                            -Dibm.jvm.sharable.application.class.path=/u/myclasses
worker.1.reset.interval=1
################################################################
# Worker 2
################################################################
worker.2.options=-Xms16M -Xmx32M -Xresettable -Xjvmset
worker.2.options.properties=-Djava.class.path=/u/myclasses
                            -Djava.compiler=NONE -Dibm.jvm.unresettable.events
                            -Dibm.jvm.sharable.application.class.path=/u/myclasses
worker.2.reset.interval=1
```

testclasses.txt

```
HelloWorld
HelloWorld
HelloWorld
HelloWorld
HelloWorld
HelloWorld
```

HelloWorld.java

```

class HelloWorld{

   public static void main(String args[]){
System.out.println("Hey!");
   }
}
```

Sample launcher for a JVMSet

Chapter 3. Performing system administration tasks

This chapter:

- Gives an overview of system administrator tasks
- Describes how classes are loaded in the Persistent Reusable JVM
- Lists useful notes for system administrators

Overview of system administrator tasks

System administration of a Persistent Reusable JVM implementation mainly concerns the management of the different types of class that can be loaded by the JVM. There are two class loaders supplied with the Persistent Reusable JVM. One loads classes as shareable trusted middleware. This is the trusted middleware class loader (TMC). The other loads classes as shareable applications. This is the shareable application class loader (SAC). The benefit in loading classes as shareable is that they then become serially reusable, that is, they do not need to be reloaded and re-JIT-compiled after each JVM-reset.

As shown in “Chapter 4. Writing middleware” on page 61, middleware is trusted in a Persistent Reusable JVM environment and can operate without restrictions. Middleware has a lifetime longer than a transaction, and can maintain state across a JVM-reset. However, applications (as shown in “Chapter 5. Developing applications” on page 75) are not trusted and must conform to a set of restrictions to ensure that they do not make the JVM unresettable. To provide isolation between transactions, applications run sequentially in a JVM that is reset each time prior to the execution of an application. A JVM-reset can only succeed if the application has not performed an unresettable action.

For the system administrator, it is suggested that access to the middleware and shareable application class repositories is controlled to prevent nonauthorized update, and to maintain any necessary auditing and backup actions. The correct class loading paths must also be specified to the Persistent Reusable JVM by the launcher at startup to ensure that the TMC and SAC are loading correctly.

The following focuses on class-loading aspects, and on what needs to be done by the system administrator to set up and maintain a Persistent Reusable JVM system.

Loading classes

The status of a class within the Persistent Reusable JVM is dictated by its class loader. For example, to be recognized as a middleware class the class must be loaded using a middleware class loader. Classes fall into five main categories:

- Primordial (system classes and standard extension classes)
- Shareable trusted middleware
- Nonshareable middleware
- Shareable application
- Nonshareable application

The Persistent Reusable JVM augments the existing Java 2 SDK, Standard Edition class-loader hierarchy to include class loaders for shareable trusted middleware and shareable applications. The class loader for nonshareable middleware classes is not supplied with the Persistent Reusable JVM, but can be supplied by a user. The class-loader hierarchy is as follows:

```
Bootstrap
     \
  Extensions
       \
    Shareable trusted middleware (*)
         \
      Shareable applications (*)
           \
        Nonshareable applications
```

where (*) indicates a class loader included with the Persistent Reusable JVM. The class-loading model is one of parent delegation: on being requested to load a class, a class loader first checks to see whether it has already loaded the class, and if not it delegates to its parent to find and load the class. Each class loader in turn going up the hierarchy looks to see whether it has loaded the class. If the class is not found by this process, each class loader attempts to find and load the class on the way back down the hierarchy. Success is achieved when the class is found and loaded by the appropriate class loader.

As with the existing (prior to the Persistent Reusable JVM) JVM, the default class loader for a running thread (that is, the so-called context class loader) is set as the default nonshareable application class loader as supplied. This allows classes of any type to be loaded, irrespective of the type of class requesting the loading. For example, a middleware class requesting a shared application class to be loaded succeeds if the class loading search is started from the context class loader (from the bottom of the hierarchy). The search proceeds up through the hierarchy looking to see if the class is already loaded, and then on the way back down the hierarchy, the shareable application class loader will succeed in finding and loading the class.

Figure 4 on page 59 shows how each category of class is loaded in the Persistent Reusable JVM.

In the Persistent Reusable JVM, system classes and standard extension classes are known collectively as primordial classes. System classes are loaded by the bootstrap class loader. Standard extension classes are loaded by the extensions class loader. Also, the bootstrap class loader and the extensions class loader are known collectively in the Persistent Reusable JVM as the primordial class loader. The default system class paths are set internally in the JVM, and point to directories relative to the path for the loaded Java library. Also, the **-Xbootclasspath** option can be used to specify the location of system classes. These classes are loaded into the system heap.

Shareable trusted middleware classes are also loaded into the system heap using the trusted middleware class loader (TMC) supplied with the Persistent Reusable JVM. The class paths to use are defined by the system property:

-Dibm.jvm.trusted.middleware.class.path=*<path>*

If this system property is not defined, the TMC is not instantiated. The TMC always loads middleware classes as shareable. System classes, standard extension classes, and shareable trusted middleware classes are loaded only once, and are never dereferenced because the primordial class loader remains active. These classes persist across a JVM-reset.

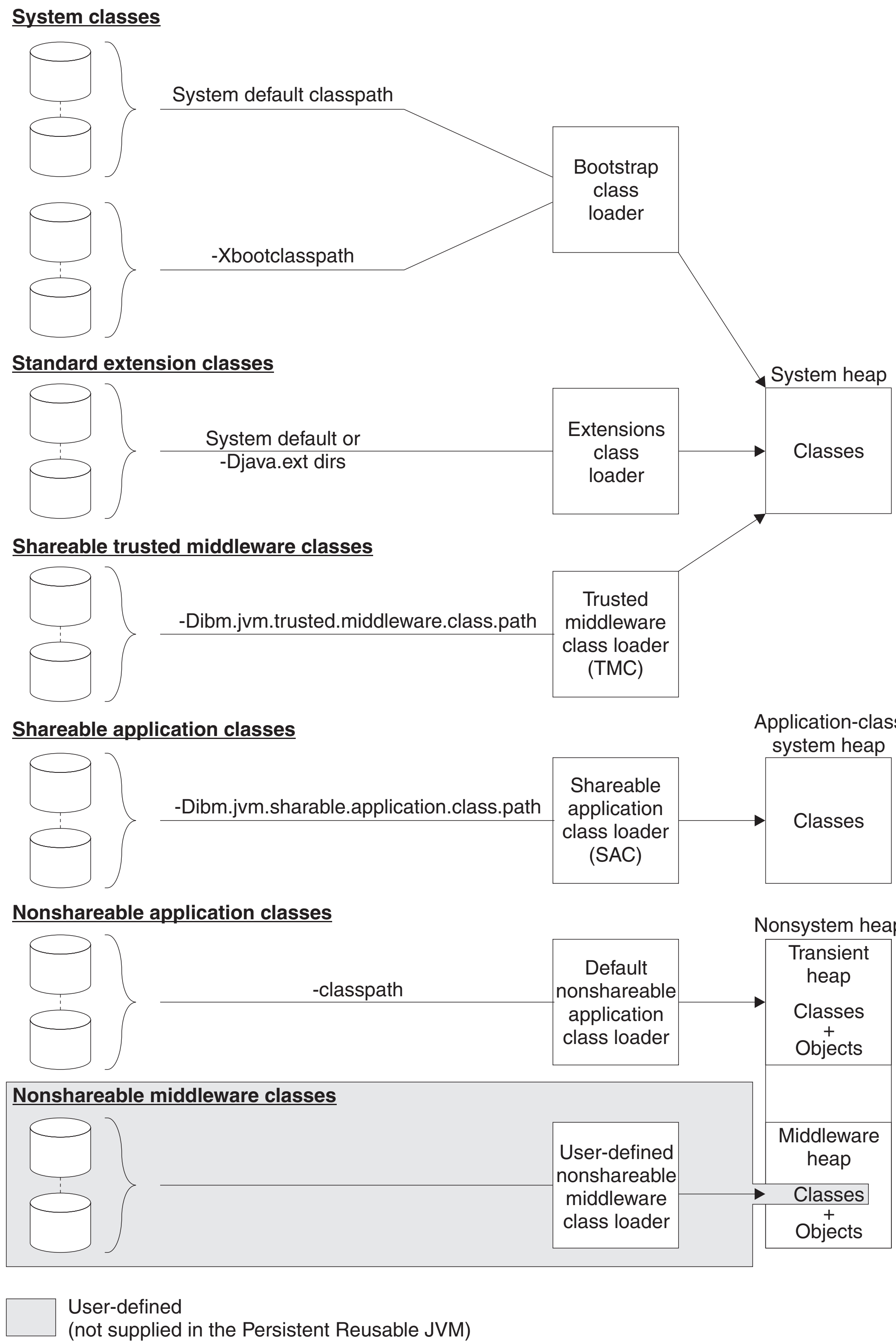

Figure 4. Class loaders and heap usage

Shareable application classes are loaded into the application-class system heap using the shareable application class loader (SAC) supplied with the Persistent Reusable JVM. The class paths to use are defined by the system property:

Loading classes

-Dibm.jvm.shareable.application.class.path=*<path>*

If this system property is not defined, the SAC is not instantiated. The SAC always loads application classes as shareable. Shareable application classes are loaded only once. They become unreachable at the end of each transaction, although the bytecodes are cached in the application-class system heap. During JVM-reset the classes are reset, and this ensures that their static initializers will rerun on first reuse. This is equivalent to reloading the classes.

Nonshareable application classes are loaded for each use of the JVM into the transient heap using the default nonshareable application class loader. The transient heap forms part of the nonsystem heap, which is a preallocated, contiguous region of storage. These classes are loaded using the CLASSPATH environment variable or the **-classpath** option. Nonshareable application classes and objects are discarded at JVM-reset because they are no longer in use.

Any nonshareable middleware classes are loaded into the middleware heap using a user-supplied, nonshareable middleware class loader. This class loading option is shown in Figure 4 on page 59 as user-defined because a nonshareable middleware class loader is not supplied with the Persistent Reusable JVM. For details on how to write one, see “Creating a class loader” on page 71.

Notes for administrators

The following points are relevant to managing a Persistent Reusable JVM implementation:

- Loading application classes as shareable in the Persistent Reusable JVM provides a performance boost in **-Xresettable** mode because the classes are serially reusable, that is, they do not have to be reloaded after each reset of the JVM. This avoids disk accesses and the need to re-JIT-compile.

  It is anticipated that most, if not all, application classes will be loaded as shareable using the SAC loader. Whether a class is loaded as shareable or not is purely a function of which class loader is used to load it - nothing must be done when writing the class in Java.
- It is important from a class loading aspect that unique class paths are specified for each of the nonsystem types of class that can be loaded by the Persistent Reusable JVM. This is achieved if middleware classes, shareable application classes, and nonshareable application classes are all in separate directories or Jar files.
- The class-path system properties for middleware and shareable applications are supplied to the JVM at startup. For the Persistent Reusable JVM, supply them as options every time a JVM is created.
- If a new version of a shareable class needs to be loaded to replace an existing version in the system heap, the Persistent Reusable JVM must be terminated and restarted.
- The reason why a JVM becomes unresettable can be determined by enabling the logging of unresettable events on the JVM that fails. This is described in “Chapter 6. Logging events” on page 81.

Chapter 4. Writing middleware

This chapter describes how to write trusted middleware for the Persistent Reusable JVM. The following sections:

- Overview of writing middleware
- Defining and loading middleware classes
- Unresettable actions for middleware
- Tidy-Up and Reinitialize methods

describe how middleware classes are tidied-up and reinitialized, allowing a Persistent Reusable JVM to be resettable and reusable. And the remaining sections:

- Considerations for middleware developers
- Creating a class loader

discuss the programming considerations in writing middleware, and describe how to write a class loader.

Overview of writing middleware

Trusted middleware classes in the Persistent Reusable JVM are privileged classes that are able to operate without the restrictions imposed on nontrusted application classes such as EJB beans or servlets. For example, they are trusted to load and manage the resettability state of native routines which cannot be checked at runtime for their resettability characteristics. Also, as they can have a persistent state across a JVM-reset, they can act as a long-term cache for middleware objects, which might provide optimizations to applications.

Trusted middleware classes remain persistent across a JVM-reset, and so do not need to be reloaded for each transaction. They manage their resettability state by:

- Tidying-up resources acquired while running the application code for a transaction when the application code completes
- Ensuring that any classes that were used by the application code for a transaction are reinitialized before reuse by the next transaction following a JVM-reset

| To achieve serial reusability, trusted middleware classes use two methods:

- `ibmJVMTidyUp()`
- `ibmJVMReinitialize()`

These Tidy-Up and Reinitialize methods are defined in "Appendix C. Application Programming Interface" on page 97.

All middleware classes referenced during the execution of a transaction application have their Tidy-Up methods called, if they exist, after the application has terminated. This is carried out during the call to `ResetJavaVM()`. Reinitialize methods are called when each middleware class is rereferenced in the next transaction, that is, after the `ResetJavaVM()` phase has completed. Specifically, a Reinitialize method call is triggered when one of the following actions is attempted for the first time on reuse:

- Create a class instance
- Invoke an instance method

- Get an instance field
- Set an instance field
- Invoke a static method
- Get a static field
- Set a static field

Both Tidy-Up and Reinitialize methods are optional; a middleware class can have either, neither, or both, specified. The following discusses the rationale for using these methods.

Tidy-Up methods should be used to perform two types of cleanup activity: transaction-related activity and JVM-related activity. Transaction-related activities tidy-up any resources that were acquired during the execution of the transaction, for example, storage, execution threads, and so on. JVM-related activities null-out any references to application objects and to application class loaders, so that the subsequent JVM-reset does not fail because garbage collection is unable to perform the `ResetGC()`function (which resets the transient heap).

Reinitialize methods can be used very much like class static initializers to perform tasks to initialize trusted middleware classes prior to their use, for example, allocating resources on a one-time basis per transaction. This activity would be performed the very first time, in a new JVM, by the class static initializers. Subsequently, as the JVM is reset and reused, the Reinitialize methods take over the task. One scenario might be that a class static initializer calls the Reinitialize method to perform all the necessary initialization actions.

A Tidy-Up method is guaranteed to be called if a middleware class is used, whereas a Reinitialize method is called only when the middleware class is rereferenced. Tidy-Up methods should therefore perform all mandatory tidy-up actions that must be executed before the next transaction is started. For example, this might include resetting any static variables that must be reset before the next transaction starts. Reinitialize methods should perform only those initialization tasks that must be executed when a class is reused.

If a trusted middleware class does not allocate any resources on a transaction basis, and does not make any references to application objects, it does not need to invoke a Tidy-Up method. For example, trusted middleware is allowed to execute native methods, and to open files and sockets and keep them open across transactions. These activities would typically not need to be tidied-up. Similarly, a Reinitialize method is only necessary if initialization tasks have to be performed when the class is rereferenced, for example, setting the time of day of the new transaction to be run. Finally, both Tidy-Up and Reinitialize methods are independent of each other, so it is possible to have a Reinitialize method without a Tidy-Up method.

At this point it is worth clearing up any possible confusion over finalizer methods and Tidy-Up and Reinitialize methods. Finalizers that have been specified for objects are queued to the finalizer thread for execution when the objects are garbage-collected. In general, a user does not know when this is going to take place, and cannot control the security context in force at the time. (This is not true for application finalizers. See "Using finalizers" on page 68.) For these reasons, it is recommended that finalizer methods are not used as a way of cleaning up resources on objects when they are garbage-collected. Tidy-Up methods are called prior to garbage collection during JVM reset, and these methods should be used to do any necessary cleanup.

For JVMs that are not reusable, that is, JVMs for which the **-Xresettable** option was not specified, trusted middleware is not relevant as Tidy-Up and Reinitialize methods are never called. Middleware classes can still be loaded by the middleware class loader; in this case, object instances go into the middleware heap along with all application objects because the transient heap does not exist.

The factors to consider in the design of trusted middleware to run in a Persistent Reusable JVM environment are as follows:

- How are middleware classes defined and loaded?
- What actions should be avoided so that JVM resettability is not compromised?
- In a Persistent Reusable JVM environment, what state persists across the JVM-reset cycle, and how is this handled by Tidy-Up and Reinitialize methods?

Defining and loading middleware classes

Trusted middleware is identified in the JVM by being loaded by a class loader tagged with the Middleware interface. The class loader supplied with the Persistent Reusable JVM, the TMC, loads these classes as shareable, that is, they are loaded into the system heap rather than the middleware heap. Alternatively, middleware writers can provide their own class loader. The process to do this is described in “Creating a class loader” on page 71.

Details of how to specify the middleware class loading path for the supplied TMC are described in “Chapter 3. Performing system administration tasks” on page 57.

Unresettable actions for middleware

Middleware classes are allowed to perform any of the unresettable actions listed in “Unresettable actions” on page 75, except the loading of nonchecked standard extension classes. Standard extensions have to be treated as primordial classes for class loading reasons, but they might contain writable statics. Unless they have been analyzed and confirmed to be free of writable statics, the JVM must treat them like any other unknown external. That is, as having the potential to cause a JVM to become unresettable.

For details on how a checked extension is loaded, see “Chapter 8. Using checked standard extensions” on page 89.

Tidy-Up and Reinitialize methods

Trusted middleware classes that need to be reset when the JVM is reused must implement one or both of the following middleware methods:

```
private static boolean ibmJVMTidyUp();
private static void ibmJVMReinitialize();
```

The methods are declared *private* so that they cannot be called directly by application code. The methods are called from within the JVM. The method names must not clash with any names that can be inherited by middleware classes or implemented as part of an interface. Hence the “ibmJVM” prefix.

These methods are not synchronized because their invocation is controlled by the Persistent Reusable JVM code, which ensures correct synchronization is enforced by acquiring the relevant object lock prior to their invocation. They can be implemented as native methods if required.

Tidy-Up and Reinitialize methods

The methods are part of the middleware class so they have the same scope for performing what would normally be unresettable actions.

Actions for Tidy-Up methods

If the following actions are not performed by the usual end-of-transaction processing, they must be carried out by middleware Tidy-Up methods:

- Close all application-specific Remote Method Invocation (RMI) connections (that is, anything that requires user access control) that were created by the middleware on behalf of the application.
- All middleware references to application class loaders and objects must be dereferenced or set to NULL, otherwise the JVM is marked as unresettable. Setting to NULL is recommended as the performance of `ResetJavaVM()` is degraded by only dereferencing them. Middleware references include middleware static variable references, JNI global references, and JNI weak references.
- All references to application-specific middleware objects must also be set to NULL.
- Free application-specific memory that was malloc'd for the transaction application by the middleware native code, otherwise a memory leak within the client JVM can occur. A memory leak within a single-client JVM could consume all the memory in the address space. Other transaction-specific resources obtained by the middleware JNI services must also be released.
- Any JVM statics or system properties that were changed by middleware must be restored to either their original values or values determined by middleware to be acceptable for subsequent transaction processing (see “Accessing JDK static variables” on page 69).
- If the middleware has created threads, ensure that only one thread exists at the termination of all the Tidy-Up calls. If multiple threads are running after all the Tidy-Up methods have been run, the JVM is marked unresettable. It is recommended to use the `thread.join()` system call to determine when threads have terminated.
- The Tidy-Up return code must be set to false whenever the tidy-up actions did not succeed.

Considerations for middleware developers

The performance gains possible with the Persistent Reusable JVM technology are only available if the authors of middleware and application code are prepared to accept some constraints on their programming practice. These constraints have been minimized but it is important to understand the choices that have been made during the Persistent Reusable JVM development. An understanding of these choices allows middleware authors to choose design options which best exploit the underlying JVM.

Maintaining resettability

It is important that application classes do not perform actions that make the JVM unresettable, and thus imply a recycling of the JVM. The actions that make a JVM unresettable are listed in “Unresettable actions” on page 75. If an application class wants to perform one of these actions, for example, set the date or time, middleware provides an appropriate wrapper to allow the application to perform the action. The assumption is that middleware performs whatever control and cleanup is necessary to restore the JVM state to an acceptable value prior to the next transaction.

Nonsystem heap management

The nonsystem heap is the area of contiguous memory that comprises both the middleware and transient heaps. The middleware heap starts from the low end of the region and grows up; the transient heap starts from the high end and grows down.

The main assumption is that middleware acts as a long-term cache (that is, longer than a single transaction) for middleware objects. These objects are created on first use of a given middleware class, for example, in response to the first use of a particular transaction. The assumption is that instances of application classes are created during a transaction, and that these instances are assigned to the transient heap. Cleanup of that heap is very efficient. It is expected that instances of transient application classes contain references to instances in the middleware heap. It is not intended that the reverse is true. Any references from middleware objects to transient application objects must be handled very carefully by the middleware classes. The ideal approach is to design the middleware classes to avoid the need for such references; for example, by copying data which is to have long-lifetime and dereferencing the original reference to the object in the transient heap. An alternative is to provide an appropriate "reset" method on the class that is called at the end of the transaction. Because a reset method is static, this approach implies the need to find all middleware objects that might contain references to transient application instances. These must all be handled at the end of a transaction. The simplest handling is to null the references. If these references are not cleared at the end of the normal middleware processing by a reset method, a Tidy-Up method must be used to null out the references.

Heap allocation and growth policy

Figure 5 on page 66 shows the initial size of each heap as defined by the associated allocation-size parameter, and shows how the heaps expand.

The initial size of the system heap is controlled by the **-Xinitsh** parameter. If this is not specified, a platform-dependent value is used. For OS/390, the default is 128 KB. Expansion by discontiguous extensions is limited only by available storage.

The initial size of the application-class system heap is controlled by the **-Xinitacsh** parameter. If this is not specified, a platform-dependent value is used. For OS/390, the default is 128 KB. Expansion by discontiguous extensions is also limited only by available storage.

The initial size of the nonsystem heap can be set by the **-Xmx** parameter. If this is not specified, a platform-dependent value is used. For OS/390, this is 64 MB. The initial size of the middleware heap within the nonsystem heap can be set by the **-Xms** parameter. If this is not specified, a value of half the platform-dependent default value is used (for OS/390 this is 500 KB - half the default value of 1 MB). The initial size of the transient heap within the nonsystem heap can be set by the **-Xinitth** parameter. If this is not specified and **-Xms** is, the initial size is taken to be half the **-Xms** value. If **-Xms** is not specified, a value of half the platform-dependent default value is used (for OS/390 this is 500 KB - half the default value of 1 MB). Expansion of the transient heap and the middleware heap within the nonsystem heap is explained in "Growth policy of the nonsystem heap" on page 66.

For a description of the JVM options, see "Command-line options, JVM options, and system properties" on page 12.

Considerations for middleware developers

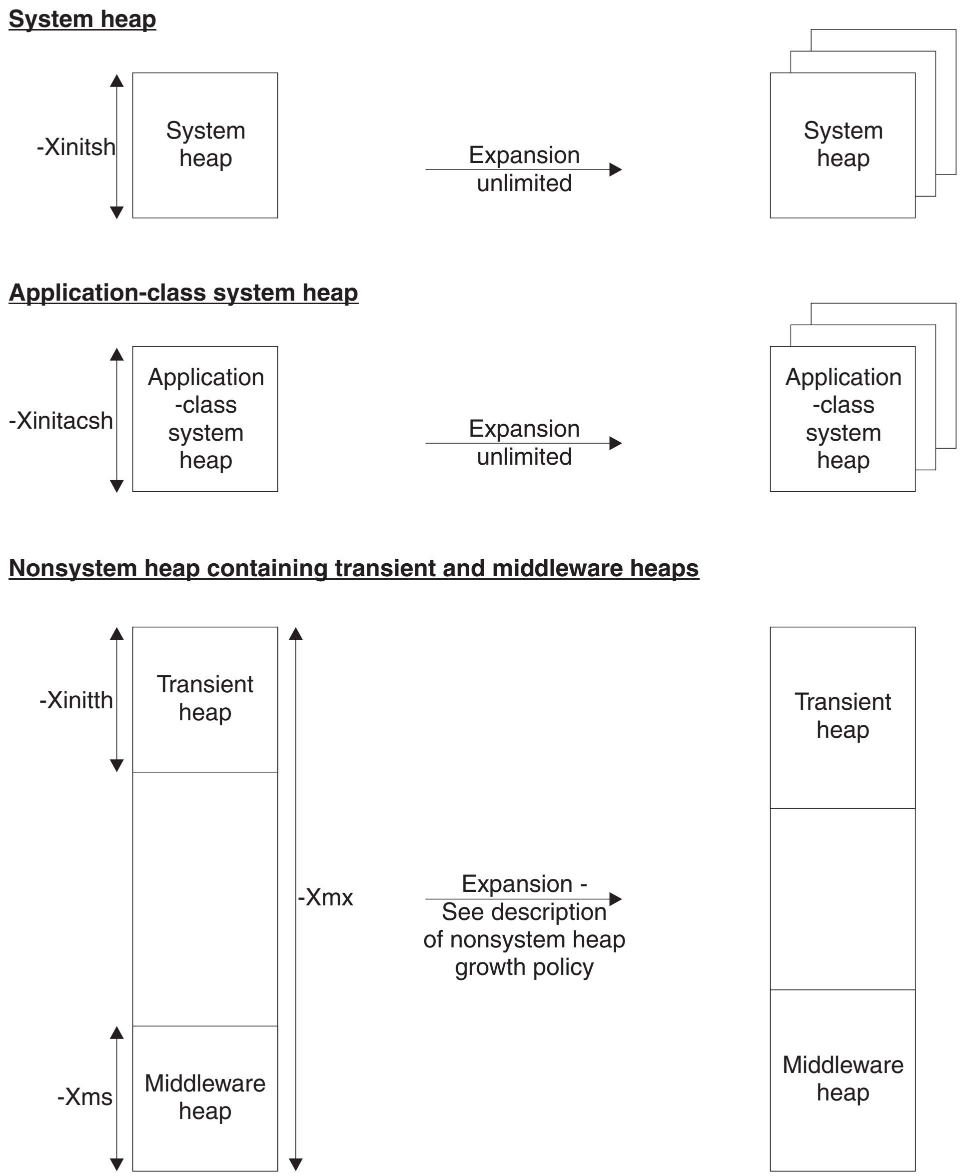

Figure 5. Heaps and how they expand

Growth policy of the nonsystem heap

The heap growth policy has been designed to avoid garbage collection. The heap growth mechanism is the same for both middleware and transient heaps and can be summarized as follows.

For a Persistent Reusable JVM, the values for **-Xmine** (minimum expansion size of the middleware and transient heaps) and **-Xmaxe** (maximum expansion size of the middleware and transient heaps) are split between the middleware and transient heaps. For example, if **-Xmine** is set to 2 MB, the minimum expansion value for both the middleware and transient heaps is 1 MB.

When heap expansion is required on either the middleware or transient heaps, a target expansion value is calculated which would provide an additional space defined by the minHeapFreePercent parameter. This can be set by the **-Xminf** option, and defaults to 0.3 or 30%. This target expansion value is then modified to be no greater than the maxHeapExpansion value if this has been specified using the **-Xmaxe** option. This modified target expansion value is then checked to ensure it is at least the minHeapExpansion size as specified using the **-Xmine** option. Finally, this modified expansion size is checked to make sure it is at least large enough to satisfy the allocation request. If insufficient space is available to satisfy the allocation request, a garbage-collection cycle is triggered on both the middleware and transient heaps. If insufficient space still exists for a middleware allocation request, an attempt is made to shrink the transient heap to a size less than the initial transient heap size. If this fails to release enough storage for the allocation request, an “out of memory” condition is returned from the storage allocation.

If the transient heap has been shrunk to less than the initial transient heap size, at reset an attempt is made to restore it to the initial size by shrinking the middleware heap and expanding the transient heap. If this fails, the JVM is marked unresettable.

For a description of the JVM options, see “Command-line options, JVM options, and system properties” on page 12.

Usage policy of the nonsystem heap

When running in resettable mode, instances of objects are allocated in either the middleware heap or the transient heap as follows:

Middleware:

- Shareable middleware object instances and nonshareable class and object instances
- Primordial object instances created in middleware context
- String objects and arrays for strings interned in the Interned String table

Transient:

- Application class and object instances
- Primordial object instances created in application context

Tracking thread context

There are two situations within the Persistent Reusable JVM when it is necessary to determine whether a thread is performing middleware or application code:

- When allocating an array or an instance of a primordial class
- When performing any action which would mark the JVM as unresettable if carried out by application code

The designation of a thread as application or middleware is known as its method_type context. This is determined as follows:

- If a primordial class method is invoked, method_type is unchanged
- If an application class method is invoked, method_type becomes application
- If a middleware class method is invoked, method_type becomes middleware
- At method return, the method_type reverts to its value at the time the method was invoked

Considerations for middleware developers

General heap usage

To investigate heap usage, enable the JVM option **-verbosegc**.

Note: Specialist knowledge of Java heaps is required to interpret the output.

Using finalizers

> **Recommendation**
> Avoid using finalizers if performance is an issue. Finalization of objects increases the JVM-reset time significantly.

The preferred approach is to avoid finalizers by using specific reset methods which are called at the end of a transaction. This approach avoids any degradation in transaction performance because of garbage-collection overheads, and avoids any security exposures that might exist because of finalization occurring under unknown security contexts (as described in the following).

The finalization of middleware objects in the Persistent Reusable JVM follows Java 2 SDK, Standard Edition design, whereby any middleware finalization objects created as a result of garbage collection are queued to run on the finalizer thread. The security context for the finalizer thread is the default that existed when the JVM was created, and so is unrelated to any security context established by the transaction application code which created the objects to be finalized. As a result, this creates potential problems for middleware object finalization. This is because the security context at finalization might not be that used to create the objects.

The finalization of application objects in the Persistent Reusable JVM does not follow Java 2 SDK, Standard Edition design. The difference in implementation is to force finalizers for application objects to run on the main thread during the JVM-reset phase, rather than on the finalizer thread. This ensures that the security context used to finalize the objects is the same as that used to create them. A disadvantage of not running finalizers until the end of a transaction is that a transaction that causes garbage collection will not be able to collect all the garbage. This is because the JVM retains the finalizer objects (and all objects reachable from them) until JVM-reset time, at which point the application has finished.

Notes:

1. This also means that any application objects that survive the reset and get promoted to the middleware heap are finalized at a later time, and so in a different security context. Such promotion is a rare occurrence, and is caused by an object with a dependency on a primordial static being allocated in the transient heap (see "Chapter 7. Processing and debugging reset trace events" on page 85 for more information). These objects are found during the garbage-collection phase of JVM-reset, where the action is to promote them to the middleware heap. This preserves the objects after the transient heap is reset.
2. Using an "empty" finalize method "{}" eliminates any undesired finalization behavior in a super class that codes a finalize method. This is because the empty finalize method overwrites the finalize method for the super class.

Accessing middleware static variables

Every middleware component must be able to reliably reset its own static state, and this can only be done by maintaining control of its static state. This means allowing access to its static variables only through getter/setter methods, that is, a middleware component *must not* expose any static variables so that they can be changed by application code directly.

Accessing JDK static variables

Because middleware is privileged to make changes to static data in the Persistent Reusable JVM without causing the Persistent Reusable JVM to be unresettable, there is the concern that independently-written middleware components will make conflicting changes. To avoid this, it is recommended that the launcher and its associated middleware code take responsibility for allowing access to JVM statics.

Avoiding nonchecked extensions

Avoid loading nonchecked extension classes as these classes could contain writable statics not known to the middleware, and can therefore not be reset during tidy-up. For this reason, loading a nonchecked standard extension class makes the JVM unresettable.

Improving efficiency in the reset-JVM loop

During JVM-reset, a flag is set as soon as the JVM is known to be unresettable. As Tidy-Up methods continue to be called it might be beneficial in the Tidy-Up methods to examine the status of this flag. Perhaps a saving can be made by avoiding some tidy-up operations that need not be carried out because the JVM will be destroyed.

The unresettable flag is accessed using a new, specific class in the Persistent Reusable JVM called:

com.ibm.jvm.ExtendedSystem

The method to access the flag is:

```
public static native boolean isJVMUnresettable()
```

This returns true if the JVM is unresettable; false otherwise.

Note: As this is a specific class in the Persistent Reusable JVM, any Java code using it will not build with another JVM.

The following is an example of how to call the `isJVMUnresettable()` method from launcher code:

```
jclass        extSys;

/* Find ExtendedSystem class */
extSys = (*env)->FindClass(env, eToUTF("com/ibm/jvm/ExtendedSystem"));
if (!extSys)
   printf("Couldn't find com.ibm.jvm.ExtendedSystem class\n");
else
{
   printf("Found com.ibm.jvm.ExtendedSystem class\n");
   /* Find the isJVMUnresettable method */
   jrmid = (*env)->GetStaticMethodID(env, extSys,
   eToUTF("isJVMUnresettable"),eToUTF("()Z") );
   if (!jrmid)
      printf("GetStaticMethodID failed for isJVMUnresettable\n");
   else
   {
      rc=printf("invoking isJVMUnresettable method\n");
      /* Call isJVMUnresettable */
           if ( (*env)->CallStaticBooleanMethod(env, extSys, jrmid) )
                   printf("JVM is unresettable\n");
           else
                   printf("JVM is resettable\n");
   }
}
```

Loading application classes

Middleware in general cannot simply use the “new” operator to instantiate application objects. This is because both the default nonshareable application class loader and Persistent Reusable JVM-supplied SAC are lower than any middleware class loaders in the class loader hierarchy. Typically, middleware should use the thread-context class loader to explicitly load an application class because this always points to the bottom of the class loader hierarchy. This class can then be used to instantiate application objects.

Memory leaks in the reset-JVM loop

It is important to ensure that all local references that are created within a reset loop are deleted using DeleteLocalRef within the launcher C-code loop. A missed reference results in a new reference being acquired each time around the loop; this causes a memory leak and increasingly longer garbage-collection and JVM-reset times. This is a significant problem when the loop executes thousands of times!

The following example, found during development, shows how easy it is to miss accumulating a local reference. Consider the following piece of launcher code:

```
str_array =
  (*env)->NewObjectArray(env, 1,
    (*env)->FindClass(env, javaString),
    jstr);
```

Every time this statement is executed a new local reference for the string class is created. As there is no variable to hold this reference it cannot be subsequently dereferenced. This can be avoided by using a variable to hold the string class reference and using DeleteLocalRef to delete the reference after it has been used:

```
stringCid = (*env)->FindClass(env, javaString);
str_array =
  (*env)->NewObjectArray(env, 1,
    stringCid,
```

```
    jstr);
  .....
  .....
(*env)->DeleteLocalRef(stringCid);
```

Alternatively, in this example the FindClass statement could be repositioned outside the reset loop and deleted after the reset loop, allowing the same reference to be reused within the loop.

New objects being created

readObject calls effectively instantiate a new object to which normal allocation rules apply (see “Usage policy of the nonsystem heap” on page 67). For example, if an application method calls readObject for a middleware object, the result object will be in the transient heap. Also, `xxx.toByteArray()` or `xxx.toCharArray()` are often used to set instance arrays. These methods generate a new array which is allocated according to normal rules. This is also true of `xxx.getBitString(security)`.

Creating a class loader

The TMC and SAC class loaders supplied with the Persistent Reusable JVM use a CLASSPATH-like setting to determine where to look for classes and Jar files. Customized class loaders can either copy this technique or use one of their own, for example, by implementing a dynamic search path or loading classes from nonstandard files.

Custom class loaders for use with Persistent Reusable JVMs must conform with the Java 2 SDK, Standard Edition class-loading specification and security mechanisms. This means that they must extend either **java.security.SecureClassLoader** or **java.net.URLClassLoader**. To preserve the class-loading delegation behavior required by the Persistent Reusable JVM, custom class loaders must also override the `findClass()` method. Take care when overriding other methods in these classes (**java.security.SecureClassLoader** or **java.net.URLClassLoader**) to avoid changing the existing behavior.

Note: JDK 1.1 style class loaders will not work with the Persistent Reusable JVM.

A custom application class-loader object must be created for each use of a Persistent Reusable JVM. If this class loader is tagged to be a shareable class loader, it must also be registered before first use using the **com.ibm.jvm.ExtendedSystem.**`registerShareableClassLoader()` API. See “Registering shareable class loaders” on page 73.

The Persistent Reusable JVM provides functionality for registering multiple shareable class loaders. This allows middleware to create multiple shareable class loaders of the same type (that is, class name) that load classes into a partitioned area of the system heap. This partitioned area is referred to in the Persistent Reusable JVM as a class loading namespace, and provides uniqueness for classes loaded by sharable class loaders. The JVM uses these namespaces to share classes across a Persistent Reusable JVM.

All known application class loaders are dereferenced during `ResetJavaVM()`, and the context class loader is set to null regardless of whether this has been set by the middleware. It is the responsibility of trusted middleware to dereference any application class loaders it has created. This allows application objects to be discarded and classes reused. New class loaders are then instantiated.

Adding a custom class loader

To add a custom class loader to the Persistent Reusable JVM, the following actions must be taken:

- Decide which of the public Java-supplied class loaders should be subclassed. As described, this must either be **java.security.SecureClassLoader** or **java.net.URLClassLoader**.
- Use the appropriate Shareable and Middleware interface tags. See "Tagging interfaces" on page 73 if the class loader is to acquire the shareable or middleware attributes.
- Decide where to attach the new class loader to the Persistent Reusable JVM class-loader hierarchy. It is not possible to insert a new class loader into this hierarchy. The appropriate parent for the new class loader is specified at its creation time. There is no API to query and return the ClassLoader object for the TMC or the SAC class loaders. This can be achieved by using the `Thread.currentThread().getContextClassLoader()` method to get the nonshareable application-class loader object at the bottom of the class loader hierarchy, and then using successive calls to the `ClassLoader.getParent()` method to work up through the hierarchy to find the SAC and the TMC as required.
- If the new class loader is to be treated as middleware by the Persistent Reusable JVM, its class must be loaded by the existing TMC loader.
- If a custom middleware class loader loads application classes, a custom application class loader must be attached as a child to the custom middleware class loader. This is illustrated in the following example:

```
                     X Bootstrap class loader
                      \
                       X Extensions class loader
                        \
                         X  Trusted middleware class loader (TMC)
                        / \
  Custom middleware    X   X  Shareable application class loader (SAC)
    class loader      /     \
 Custom application  X       X  Nonshareable application class loader &
    class loader                Context class loader
```

If required, a custom application class loader alone can be installed; similarly with either the TMC or SAC as its parent. For example:

```
                     X Bootstrap class loader
                      \
                       X Extensions class loader
                        \
                         X Trusted middleware class loader (TMC)
                          \
                           X  Shareable application class loader (SAC)
                          / \
 Customer application    X   X  Nonshareable application class loader &
     class loader               Context class loader
```

General rules

Rules concerning custom class loaders are as follows:

- The parent of a shareable class loader must also be shareable, for example, the TMC or the primordial class loader.

- The parent of a middleware class loader must either be a middleware class loader, for example, the TMC, or the primordial class loader.
- Middleware class loaders must be middleware objects, that is, typically loaded and created by middleware.
- An application class loader should typically be created with a middleware class loader as its parent. This will ensure that any middleware classes on which applications have a dependency can be resolved.
- Typically, all custom class loaders should be middleware objects, especially if a security manager is going to be employed. This is because class loaders need read access to the classes they are trying to load while not giving applications read access to these classes.
- Some Java exceptions thrown because of class loader errors might result in control being returned to the JVM launcher; typically the C program that called `JNI_CreateJavaVM()`. So it is essential that either the `ExceptionCheck()` or `ExceptionOccured()` JNI API be used after running a Java program. A full description of the exception and related stack trace is available.
- It is recommended that middleware writers concerned with class loading, whether using custom class loaders or not, consult the *Inside Java 2 Platform Security: Architecture, API Design, and Implementation* book (or a similar publication) for background information on the topic.

Tagging interfaces

Two tagging interfaces (like java.io.Serializable) identify shareable and middleware class loaders. They are used by the supplied TMC and SAC loaders and by a modified standard extensions class loader. Middleware providers wanting to supply their own class loaders must use them to get the required attributes. The interfaces have no methods.

The tagging interfaces are:

```
package com.ibm.jvm.classloader;

public interface Middleware
{
}

package com.ibm.jvm.classloader;

public interface Shareable
{
}
```

Implementing either, or both, triggers the appropriate class behavior in the JVM. Middleware class loaders will probably want both. No functionality is required in the class loader itself; it should load the classes according to the required algorithm.

Registering shareable class loaders

You register a shareable class loader by using a static Java method, **com.ibm.jvm.ExtendedSystem.**`registerShareableClassLoader()` (see “Appendix C. Application Programming Interface” on page 97). This method must be called either from the constructor of a shareable class loader or explicitly by middleware after creating the shareable class loader, but before using it.

Note: It is necessary to create and register a new shareable application class loader on each run of a Persistent Reusable JVM.

Creating a class loader

The `registerShareableClassLoader()` method can throw the following exceptions:

- public class **NamespaceInUseException** extends **NamespaceException**
- public class **ClassLoaderAlreadyRegisteredException** extends **NamespaceException**
- public class **ClassLoaderParentMismatchException** extends **NamespaceException**
- public class **InvalidClassLoaderParentException** extends **RuntimeException**
- public class **InvalidMiddlewareClassLoaderException** extends **RuntimeException**

These exceptions are described in "Appendix C. Application Programming Interface" on page 97.

Note: The first three exceptions are all children of the parent exception class **NamespaceException**, which is provided to simplify calls to the registration API, and allows the middleware provider to code a single "catch block" to handle registration exceptions.

Chapter 5. Developing applications

This chapter:

- Explains how a JVM is made unresettable
- Describes the actions that make a JVM unresettable
- Describes the conditions that make a JVM unresettable
- Describes how an application can update a static variable, and lists the Java methods that update static variables
- Gives tips for application developers

How a JVM is made unresettable

Application classes execute particular business functions, either as simple classes or as EJB beans. Application classes might use underlying middleware to update data in enterprise systems (such as transaction monitors or databases), or they might be self-contained. Such application code can be written in-house or can be bought from a vendor. Application code is expected to hold data pertaining only to the transaction being run. It is not expected to have any knowledge of reusability, and is not expected to perform any tidying-up.

Application classes are not trusted and must follow a strict set of rules to avoid making the JVM unresettable. Restricted actions are not prevented, but if performed by the application they can potentially leave the JVM in an undefined state when the application is terminated. The application cannot be relied upon to restore this state, and the JVM is not allowed to restore state on behalf of the application as this would break Java compliance rules. This is why such actions are unresettable actions, and why application code is untrustworthy.

Unresettable events and conditions are checked for when a JVM is marked reusable at startup using the **-Xresettable** option. If an unresettable action is detected during the execution of the application, the JVM is marked unresettable and the launching subsystem destroys and re-creates the JVM.

Unresettable actions

The following unresettable actions apply only to application classes. If an application class performs one of these actions, the JVM becomes unresettable and is destroyed at the end of the current transaction. Trusted middleware classes can perform any of these actions, except the loading of a nonchecked standard extension class. If the unresettable action in the following list is also an EJB bean restriction, this is indicated in the action heading. The reason codes for each unresettable action are listed in “Appendix A. Unresettable reason codes” on page 91.

Unresettable actions

- Writing to a static variable associated with a class loaded by the primordial class loader.

  Note: The JVM does not reset writable static variables or rerun the static initializers for the primordial classes because this would break compliance. The methods that either modify or access static variables are listed in "Java methods that make the JVM unresettable" on page 78.
- Using the Reflection API to modify the accessibility of fields.

  Using the `java.lang.reflect.AccessibleObject.setAccessible()` method causes the JVM to become unresettable. An application could use this method to gain access to a private static variable directly, circumventing the unresettable action checks.
- Setting System properties. (EJB)

  Methods `java.lang.System.setProperty()`, `java.lang.System.setProperties()`, and `java.lang.System.getProperties()` cause the JVM to become unresettable. `java.lang.System.getProperties()` is restricted because it returns the java.util.Properties object that stores the system properties; once you have access to this, you can set system properties using, for example, `java.util.Properties.load()`.

  As an application developer, if you want to access the system properties without making the JVM unresettable, you can use the `java.lang.System.getProperty()` method. This returns the value of a property as a String, which does not allow you to then modify it.
- Using the Abstract Windowing Toolkit (AWT) or any of its derivatives to access a display or keyboard, or to print, as this could affect the behavior of the next application. (EJB - except printing) This unresettable action also applies to middleware.
- Setting the context class loader or security manager. (EJB)

  The following methods make the JVM unresettable:
  - `java.lang.Thread.setContextClassLoader()`
  - `java.lang.System.setSecurityManager()`
- Attempting to access or modify the security configuration objects, that is, Policy, Security, Provider, and Signer and Identity objects. (EJB)

  The following methods make the JVM unresettable:
  - `java.security.Identity.setPublicKey(), addCertificate(), setInfo()`
  - `java.security.IdentityScope.setSystemScope()`
  - `java.security.KeyStore.deleteEntry(), setCertificateEntry(), setKeyEntry()`
  - `java.security.Policy.getPolicy(), setPolicy()`
  - `java.security.SecureRandom.setSeed()`
  - `java.security.Security.insertProviderAt(), removeProvider(), setProperty()`
  - `java.security.Signature.setParameter()`
  - `java.security.Signer.setKeyPair()`
- Redirecting the input, output, or error streams as this affects the behavior of the next application. (EJB)

  Methods `java.lang.System.setErr()`, `java.lang.System.setIn()`, and `java.lang.System.setOut()` cause the JVM to become unresettable.
- Closing the standard input, output, or error streams. (EJB)

  For example, by doing: `System.in.close()`.

- Managing threads. This is to ensure that there cannot be any access to the transient heap while it is being cleared. (EJB)

  Any use of the thread methods to start, stop, resume, suspend, or interrupt threads marks the JVM unresettable.
- Loading a native library provided by the application. (EJB)

  This is because it cannot be known what native code provided by the application does. For example, it might modify primordial static variables without using the methods that include checks.

  Using the methods `java.lang.System.loadLibrary()`, `java.lang.System.load()`, `java.lang.Runtime.loadLibrary()`, or `java.lang.Runtime.load()` marks the JVM unresettable.
- Creating a new process using `Runtime.exec()`.
- Loading a nonchecked standard extension.

  Standard extensions must be treated as primordial for class loading reasons, but they might contain writable statics. Unless they have been analyzed and confirmed to be free of writable statics, the JVM must treat them like any other unknown external. That is, as having the potential to cause a JVM to become unresettable. This unresettable action also applies to middleware.
- Using the JVM in debug mode.

  Specifying the **-Xdebug** option makes the JVM unresettable. This is because the monitoring activity is performed on a separate thread that is asynchronous to the main Java thread that calls `ResetJavaVM()`.

By default, the JVM checks to see if the application carries out any of the unresettable actions listed. If it does, the JVM marks itself as unresettable but continues to run the application to completion, at which point any attempt to reset the JVM fails.

It is possible for a potentially unresettable action to be caught by the security policy; if this happens, a security exception is raised. In these cases, the security policy has prevented the action from being performed and consequently the JVM is not marked as unresettable.

Once a JVM is marked unresettable, it remains unresettable. This means that the JVM state is undetermined and that the JVM must be destroyed and re-created.

Unresettable conditions

In addition to the unresettable actions listed in "Unresettable actions" on page 75, there are also a number of unresettable conditions that are recognized by a Persistent Reusable JVM. For the complete list of unresettable actions and conditions, see "Appendix A. Unresettable reason codes" on page 91.

Modifying static variables

If an application modifies a primordial (system) static variable, the JVM must mark itself unresettable because it cannot reset the variable to its initial state during `ResetJavaVM()` without breaking compliance. An application can modify primordial static variables in a number of ways:

- Directly setting a public or protected variable.
- Using public or protected setter methods.

Modifying static variables

- Using a public or protected getter method that modifies the variable under the covers. For example, if the variable has not been initialized when the getter method is called, the method might set it to some default value.
- Using a public or protected getter method to get the object reference and then using other methods to update it. In this case, the JVM is marked as unresettable if the getter method is called, as it cannot be known if later method calls actually modify the object.

Java methods that make the JVM unresettable

The following table lists all the Java™ methods that update static variables. There are no JVM-native methods that update static variables. If application code uses one of these Java methods, the JVM is marked as unresettable.

Table 2. Java methods that update static variables

Class	Static field	Modified or accessed by
com.ibm.org.omg.SendingContext._CodeBaseImplBase	__ids	_ids
com.ibm.rmi.util.JDKBridge	localCodebase	setCodebaseProperties
java.beans.Beans	designTime guiAvailable	setDesignTime setGuiAvailable
java.beans.Introspector	searchPath	setBeanInfoSearchPath
java.beans.PropertyEditorManager	searchPath	setEditorSearchPath
java.lang.Shutdown	runFinalizersOnExit runFinalizersOnExit runFinalizersOnExit	setRunFinalizersOnExit exit Runtime.runFinalizersOnExit
java.net.Authenticator	theAuthenticator	setDefault
java.net.HttpURLConnection	followRedirects	setFollowRedirects
java.net.URLConnection	defaultAllowUserInteraction defaultUseCaches factory fileNameMap	setDefaultAllowUserInteraction setDefaultUseCaches setContentHandlerFactory setFileNameMap
java.rmi.activation.ActivationGroup	canCreate currGroup currGroupID currSystem	createGroup destroyGroup destroyGroup setSystem
java.rmi.server.LogStream	java.rmi.server.LogStream known	setDefaultStream log
java.rmi.server.RMISocketFactory	handler	setFailureHandler
java.rmi.server.RemoteServer	log	setLog
java.sql.DriverManager	drivers logStream logWriter loginTimeout	registerDriver setLogStream setLogWriter setLoginTime
java.util.Locale	defaultLocale	setDefault
java.util.TimeZone	defaultZone	setDefault

Tips for application developers

The following are tips for developing applications:

- Applications should not use the Sun packages (for example, **sun.applet**) directly, as they are not part of the supported public Java interface and might be changed without warning.

  Note: The Persistent Reusable JVM does not check for writable statics that are changed using access methods in a Sun package. It only polices the core Java packages for statics that are changed by Java.
- In general, the use of finalizers is discouraged. However, finalization of an application object in the Persistent Reusable JVM is possible without the problem of the security context of the finalization thread being different to that used when the object was created. This is achieved by queuing any application object finalizers to run at the end of the transaction application during JVM-reset, and so on the same application thread that created the objects.

  Note: Running application finalizers at JVM-reset imposes a performance penalty beyond executing the finalizer method for each object. This is because executing the finalizer method and any additional finalizers that might be recursively invoked could result in the JVM being in an unresettable state. So it is necessary to redo the checks for unresettability, such as ensuring that there are no cross-heap pointers from active middleware objects to transient heap objects. These checks can be very costly.
- There are no specific class-loading concerns that need to be addressed by the application writer. A class can be loaded either as shareable or nonshareable, depending on where it is loaded from, and this is generally controlled by the system administrator, with input from the application writer as required. For example, the class might need to be loaded as shareable so that it becomes serially reusable. For further information, see "Chapter 3. Performing system administration tasks" on page 57.

  Although the Persistent Reusable JVM supports customized user-supplied class loaders for shareable and middleware classes (see "Creating a class loader" on page 71), it is recommended that customized application class loaders should be the responsibility of middleware developers.
- Application Development Environments such as VisualAge for Java (VAJ) can be used to develop and test application classes. However, this is not necessarily true for middleware development because the Persistent Reusable JVM environment must be able to run the Tidy-Up and Reinitialize methods, if they exist.
- If an application class subclasses a middleware class it does not automatically become middleware because that is determined solely by how the new class is loaded. If it is subsequently loaded by the TMC loader, it is treated by the Persistent Reusable JVM as middleware.

Tips for application developers

Chapter 6. Logging events

This chapter describes how to enable event logging, and how to use system properties to log:

- Unresettable actions
- Reset trace events
- Cross-heap events

It also gives examples of logging output.

Enabling event logging

To log events in a resettable JVM, it is first necessary to enable general event logging in the JVM. This is done using the following system property:

-Dibm.jvm.events.output=*{<path/filename>* | *stderr* | *stdout}*

which specifies where to log the events, that is, either in a specified file or on one of the standard system output streams. When a loggable event occurs, an event record containing information describing the event (where it occurred, and why it occurred) is generated provided the system property enabling the event has been specified. The different types of event that can be logged, and the associated system property used to enable them, are described in the following sections.

Typically, an application writer tests an application with full logging, and releases the application only after ensuring that there are no unresettable actions. Similarly, if a JVM-reset fails while in operation, logging can be turned on by the support group to determine why the JVM failed a reset operation.

Logging unresettable actions

Logging of unresettable actions allows the system developer to determine why a JVM was flagged as unresettable and the subsequent `ResetJavaVM()` returned a reset-failed return code.

In normal operation (with logging disabled), when the unresettable flag is set for the first event no further checking for unresettable events is made. When logging of unresettable events is enabled, checking for these events is active continuously. This enables a developer to debug a new system and see all unresettable actions that can be generated during the transaction.

The following system property:

-Dibm.jvm.unresettable.events.level=*{min* | *max}*

enables the logging of unresettable events and specifies the level of logging detail recorded in the event record. The minimum level (min) contains only the name of the unresettable action. The maximum level (max) also includes a stack trace to help identify the offending module.

The option to output the events on one of the standard system output streams allows a transaction processing system such as CICS to merge the output from the streams and prefix messages with date, time, and system ID information. This allows a customer to correlate the JVM-reset failure with the unresettable actions that caused it and the modules where the actions occurred.

Example of logging output for unresettable actions

The first four lines of the event record are a header, and these lines appear each time the file is reused by a JVM. The actual event begins with a header [EVENT <eventnum>], and ends with [END EVENT]. Inside this are description, class thread name and thread number, and a time stamp.

```
[****************** EVENT LOG FILE HEADER *****************]
START DATE: Tue Aug 29 16:20:01 2000
MILLIS    : 480
[****************** END EVENT FILE HEADER *****************]

[EVENT      0x0000000A]
TIME=29/08/2000 at 16:22:23.588
THREAD=MyProcessingThread (0:7809792)
CLASS=UnresettableEvent
DESCRIPTION=The JVM cannot be reset because a native library was loaded
STACK=
com.appgen.MyClass.loadMyNativeCode (MyClass.java:520)
com.appgen.MyClass.Initialize(MyClass.java:420)
... (more frames not shown)
[END EVENT]
```

This syntax is used because it is easy to parse, and to present as XML or HTML.

Logging reset trace events

Reset trace events occur during the garbage collection phase of the `ResetJavaVM()` call. One of the checks that must be performed prior to resetting the transient heap is to see whether there are any live references from objects in the middleware heap to objects in the transient heap. Sometimes a reference is found from a middleware object to a transient heap object but it is unclear whether the middleware object is still active. In this case, the JVM initiates an expensive “trace-for-unresettability” check to determine whether such references are live or not. In other cases, a heap compaction might have occurred during the transaction, in which case a trace-for-unresettability check is the only way to determine whether such middleware to transient heap references exist, and are live.

Reset trace events, therefore, represent an expensive activity at JVM-reset. This can be avoided by modifying either middleware logic (to null references) or heap sizes and the expansion policy to ensure that garbage collection does not occur between resets.

The system property to enable reset trace events is:

-Dibm.jvm.resettrace.events

“Chapter 7. Processing and debugging reset trace events” on page 85 describes how to locate reset trace events and how to amend code to remove them.

Example of logging output for reset trace events

The first four lines of the event record are a header, and these lines appear each time the file is reused by a JVM. The actual event begins with a header [EVENT

<eventnum>], and ends with [END EVENT]. Inside this are description, class thread name and thread number, and a time stamp.

```
[****************** EVENT LOG FILE HEADER *****************]
START DATE: Thu Nov 09 13:21:50 2000
MILLIS    : 734
[****************** END EVENT FILE HEADER *****************]
[EVENT 0x1]
TIME=09/11/2000 at 13:21:51.046
THREAD=main (0:753110)
CLASS=ResetTraceEvent
DESCRIPTION=0x08CDBEE0 is an instance of Application from 0x00D7D668.
0x00D7D668 is an instance of Middleware
[END EVENT]
[EVENT 0x1]
TIME=09/11/2000 at 13:21:51.093
THREAD=main (0:753110)
CLASS=ResetTraceEvent
DESCRIPTION=0x08CDBEE0 is an instance of Application from 0x00D42D30.
0x00D42D30 is an instance of Middleware
[END EVENT]
```

Logging cross-heap events

At JVM-reset, a cross-heap event is generated when an object in the middleware heap is found to reference an object in the transient heap.

The system property to enable cross-heap events is:

-Dibm.jvm.crossheap.events

See “Chapter 7. Processing and debugging reset trace events” on page 85 for more information on cross-heap events.

Logging cross-heap events

Chapter 7. Processing and debugging reset trace events

This chapter describes the reasons why reset trace events occur, and provides information to assist in interpreting the event log information that can be obtained when using the debug build of the Persistent Reusable JVM.

Reset trace events in the Persistent Reusable JVM

At JVM-reset, the system heap, the middleware heap (in the nonsystem heap) and the Java stacks are scanned, and references to objects in the transient heap are noted. If there are no references, the JVM is “clean” (resettable), and the transient heap is simply removed and the reset process continues. This is the ideal situation and gives the best reset performance.

If there are references into the transient heap, it is necessary to determine whether these are still active, in which case a problem exists, or whether they are dead, in which case they will be collected when the middleware heap is garbage collected.

When a cross-heap reference is found, a “trace-for-unresettability” check is initiated which takes the known live object set and scans from these objects, via their fields, to objects that are referenced. If any of these referenced objects are found to be in the transient heap, the cross-heap reference is live, and one of two actions can take place:

- The first action is recovery (known as promotion), where the object and all objects referenced by the objects’ fields are promoted by moving them from the transient heap to the middleware heap. This can only be done if the objects are primordial (Strings, and so on), and not objects of application classes. If this process is successful, the offending cross-heap reference has been removed and the reset of the transient heap can proceed.
- The second action is to mark the JVM as unresettable, because a live reference to a transient heap application object still exists. At this point, the reset stops and an “unresettable” event is generated.

The initial aim is to have no unresettable events, which corresponds to no live links to objects in the transient heap which are, or have links to, application objects. Having achieved this, there are then possible performance improvements to make. This is where a debug build of the Persistent Reusable JVM will help.

If you do have dead links to application objects, or live links to promotable (primordial) objects, reset will not fail but it was necessary to perform the trace-for-unresettability check to determine that this was the case. This check is expensive and corresponds approximately to the first phase of garbage collection where the heap is scanned for live objects.

Efficiency is a matter of perspective. In those applications with a long life cycle, a longer JVM-reset time might be acceptable. Whereas if your transactions are short, the JVM-reset time will become a significant factor.

Debugging reset trace events

You will know that a trace-for-unresettability check has been started because the log will show an event called a ‘reset trace event’. You will only see one per JVM-reset because, as described, once a single cross-heap reference is found, the scan is abandoned, and the trace-for-unresettability check is started.

Debugging reset trace events

With a reset trace event in the event log you will want to know:

- Where in the code is the cross-heap reference created?
- Are there more reset trace events?
- What can I do about the reset trace events?

To answer these questions you will need to run the debug version of the Persistent Reusable JVM, and you will need to run with the JIT compiler turned off.

The first thing that the debug version does is to continue the scan, even if a reset trace event has occurred, so you see all the reset trace events, and you know all the cross-heap references in one pass.

The second thing the debug build does is to generate a ‘cross-heap reference event’ each time one of these references is created, that is, every memory-write is tested to see if the ‘from’ address is middleware and the ‘to’ address is transient.

There is one further event generated, which is a reset trace event for a promotion. The following explains why this event is generated.

The log that you get, therefore, contains reset trace events, cross-heap events, and promotions. There will be many cross-heap reference events which are no longer valid (that is, they have already been nulled out), so the main task is to match the address in a reset trace event (or unresettable event) with a corresponding cross-heap event. Ideally, you will want to sort this event log into groups, headed by each reset trace event, where each group lists all the cross-heap events which might be associated.

There is one case where you might see an unresettable event with no corresponding cross-heap reference event. This is where a primordial object has been promoted, but the references from it are application objects. These cannot be promoted, but the reference to them, when created, was from transient to transient, and was not cross-heap. Hence the need for promotion events. You can see that the object was promoted, and associated with the promotion event will be the original cross-heap reference which created the need to promote.

Once you have the original location of the creation of these cross-heap references, you can see if it is possible to cut the cross-heap reference at some later point. Here you need to know that it is not sufficient to cut the root of a tree of references, where one of the branches of the tree is cross-heap. This just creates a dead reference, which still invokes the reset trace. You need to cut the actual reference which goes from one heap to the other, by nulling the reference. If the reference appears to be in JVM code, IBM should be involved in the fix. Contact IBM via your normal service channel.

The event logging mechanism itself is turned on by defining **-Dibm.jvm.events.output**, and each type of event is turned on by specifying one or more of the following:

- -**Dibm.jvm.crossheap.events**
- **-Dibm.jvm.resettrace.events**
- **-Dibm.jvm.unresettable.events.level**

See “System properties” on page 19 for a description of these system properties.

Remember that the JIT compiler must be off, and that you must run the debug build to get cross-heap events, and to get more than one reset-trace event.

Likely scenarios

An unresettable event matches directly to a cross-heap reference event.
This means that there is a direct link from a live middleware object (probably a static or rooted in a static) to an instance of an application object. From the cross-heap reference stack dump, you should be able to find the link being set up, and decide the best place to cut the link before reset.

An unresettable event matches one or more promotion events, followed by a cross-heap event.
This means that a live middleware object references a primordial object in the transient heap, which got promoted to the middleware heap, but was then found to reference an application object. You need to examine where the original primordial was referenced from the middleware object, and try to cut that link before reset. If this is not possible, you need to find the final application object, and try to cut the link from the primordial to the application object. The second method will still leave promotions and reset traces, so it is not the best solution.

Note: In both of the above cases, you might find that the middleware object is not a static, but is referenced from a static. You might be able to cut the link from the static to the middleware object before reset. However, that simply means that the cross-heap reference is still there, but dead, so all the work will still need to be done at reset to resolve the events, and you will still get reset trace events.

There is no unresettable event, but a reset trace event matches a cross-heap event.
Here you will have found a cross-heap pointer and been able to establish that it is no longer rooted in anything live (and could in fact be collected). These references will not stop you resetting, but they do cause a trace-for-unresettability check of the whole heap at reset to find out if they are live. If you can remove the originating cross-heap reference before reset, your reset will perform more efficiently.

Likely scenarios

Chapter 8. Using checked standard extensions

Standard extensions are optional packages that extend the core Java platform. The Java Naming and Directory Interface (JNDI) is an example of such a package. Standard extensions are loaded, like system classes, by the primordial class loader. This presents a problem in Persistent Reusable JVMs. Primordial classes are all checked for the existence of writable statics, and checks are carried out to identify any changes that are made at execution time. However, standard extensions, by default, are not checked and can potentially cause a JVM to become unresettable. For this reason, whenever standard extensions are loaded a JVM is marked unresettable.

Checked standard extensions are those extensions that have been checked for writable statics and are known to be clean. Specifically, the writable statics are either not exposed to application code, or checks have been added to the standard extension to detect any changes that are made. Loading a checked standard extension does not cause the JVM to become unresettable.

Note: Being checked does not preclude a standard extension from making the JVM unresettable at runtime because it has performed an unresettable action.

Checked extensions are identified by the presence of a new manifest entry in the main section of the JAR file for the extension. This entry is as follows:

```
IBM-Reusable-JVM-Compatible: True
```

The extensions class loader looks for this entry at runtime. If either a manifest is absent or the new entry cannot be found, the JVM is marked unresettable with SCJVM_LOADING_UNCHECKED_EXTENSION (see “Appendix A. Unresettable reason codes” on page 91).

Appendix A. Unresettable reason codes

The following table lists the reason codes explaining why a Persistent Reusable JVM has been marked as unresettable.

| **Note:** Code 0x00000000 (SCJVM_CLEAN) indicates a resettable JVM.

Table 3. Unresettable reason codes

Code	Reason
0x00000001	SCJVM_MODIFYING_STATIC An application has updated a static variable in the JVM. This update cannot be undone at JVM-reset because this would break Java compliance. The methods that either modify or access static variables are listed in "Modifying static variables" on page 77.
0x00000002	SCJVM_AWT Middleware or application use of the AWT or any of its derivatives to access a display or keyboard, or to print, is not allowed as this could affect the next application.
0x00000003	SCJVM_MODIFYING_SECURITY Application code accessing or modifying security configuration objects such as Policy, Security, Provider, Signer, and Identity is not allowed. See "Unresettable actions" on page 75 for the methods that make the JVM unresettable.
0x00000004	SCJVM_SETTING_CLASSLOADER_OR_SECURITY_MANAGER Application code setting the context class loader or the security manager is not allowed. See "Unresettable actions" on page 75 for the methods that make the JVM unresettable.
0x00000005	SCJVM_USING_REFLECTION Modification of the accessibility of fields using the Reflection API is not allowed by application code because an application could gain access to private static variables directly, thereby circumventing the unresettable checks for these variables. See "Unresettable actions" on page 75 for the methods that make the JVM unresettable.
0x00000006	SCJVM_PROPERTIES Accessing or setting system properties by application code is not allowed. See "Unresettable actions" on page 75 for details of how to access system properties without making the JVM unresettable.
0x00000007	SCJVM_REDIRECTING_IO Redirecting the input, output, or error streams by application code is not allowed because this could affect the behavior of the next application. See "Unresettable actions" on page 75 for the methods that make the JVM unresettable.
0x00000008	SCJVM_CLOSING_IO Closing any of the standard input, output, or error streams by application code is not allowed.
0x00000009	SCJVM_THREADS Any use by application code of the thread methods to create, start, stop, resume, or suspend threads is not allowed. This is to ensure that there cannot be any access to the transient heap while it is being cleared. Setting the thread priority of the main thread is also not allowed.

Unresettable reason codes

Table 3. Unresettable reason codes (continued)

Code	Reason
0x0000000A	SCJVM_JNI Application code in a JVM is not allowed to load a native library provided by the application. This is because there are no checks on what the native library might do to the JVM, or on whether the state of the JVM is changed by the native library, thereby making it unresettable. See "Unresettable actions" on page 75 for the methods that make the JVM unresettable.
0x0000000B	SCJVM_CREATING_PROCESS Application code creating a new process is not allowed.
0x0000000C	SCJVM_TIDYUP_FAILED A middleware Tidy-Up method has failed and the JVM has been marked unresettable.
0x0000000D	SCJVM_APPLICATION_OBJECT_REACHABLE_FROM_STATIC An instance of an application class is reachable from a static variable belonging to one of the system or extension classes.
0x0000000E	SCJVM_REF_FROM_MW_TO_TH A middleware object (instance or array) contains a reference to an application object.
0x0000000F	SCJVM_MW_STATIC_VARIABLE_IN_TH A static variable belonging to a middleware class has been found to be an application object or a primordial object created by application code.
0x00000010	SCJVM_JNI_GLOBAL_REFERENCE_IN_TH A JNI global reference to an application (or application-created) object was found during the `ResetJavaVM()` call.
0x00000011	SCJVM_PINNED_OBJECT_FOUND_DURING_RESET A pinned object was found in the transient heap while trying to promote objects to the middleware heap. This is probably a program error.
0x00000012	SCJVM_TRANSIENT_HEAP_TOO_SMALL There is not enough space in the nonsystem heap, which contains both the middleware and transient heaps, to allow the transient heap to be created with the required initial size. Increase the size of the nonsystem heap using the **-Xmx** option.
0x00000013	SCJVM_MIDDLEWARE_RETAINING_APPLICATION_CLASSLOADER A middleware instance has retained a reference to an application class loader (for example, the context class loader).
0x00000014	SCJVM_LOADING_UNCHECKED_EXTENSION Loading an extension which has not been checked for writable statics is not allowed and makes the JVM unresettable. This restriction applies to application code and middleware code.
0x00000015	SCJVM_IN_DEBUG_MODE Using **-Xdebug** makes the JVM unresettable.
0x00000016	SCJVM_MULTIPLE_THREADS_AT_RESET Multiple middleware threads were detected after all the middleware Tidy-Up methods were run. Only the thread on which `ResetJavaVM` is called should be running after all Tidy-Up methods have completed.

Table 3. Unresettable reason codes (continued)

Code	Reason
0x00000017	SCJVM_REINITIALIZE_FAILED A middleware Reinitialize method has failed and the JVM has been marked as unresettable.
0x00000018	SCJVM_JVM_INTERNAL_ERROR A JVM internal error has been detected and the JVM has been marked as unresettable.
0x00000019	SCJVM_REF_TO_SHAREABLE_APPLICATION_CLASS A middleware object has retained a reference to a shareable application class.
0x0000001A	SCJVM_REF_TO_NONSHAREABLE_APPLICATION_CLASS A middleware object has retained a reference to a nonshareable application class.
0x0000001B	SCJVM_SECURITYMANAGER_NOT_PRIMORDIAL_OR_MIDDLEWARE It is not permissible to use a Security Manager which is loaded as an application class.
0x0000001C	SCJVM_PROMOTION_OUT_OF_MEMORY There is insufficient middleware heap space to perform the reset of the transient heap space.
0x0000001D	SCJVM_OUTSTANDING_JAVA_EXCEPTION The launcher has called `ResetJavaVM()` without first clearing an outstanding Java exception. The launcher must either clear the exception before calling `ResetJavaVM()` or destroy the JVM.
0x0000001E	SCJVM_TH_OR_ACSH_POINTER_FROM_LAUNCHER The launcher code has retained a reference to an application class or object.
0x0000001F	SCJVM_TH_OR_ACSH_POINTER_IN_NON_LAUNCHER C or C++ code, other than the launcher program, has retained a reference to an application class or object.

Unresettable reason codes

Appendix B. Reset trace events

The following table lists the reset trace events that can occur during JVM-reset. These events cause a time-consuming trace-for-unresettability check to occur. This trace checks that there are no references from live objects in the middleware heap to objects in the transient heap.

Table 4. Reset trace events

Code	Reason
1	RESETTRACEEVENT_REFERENCE_TO_TH A reference was found from an object in the middleware heap to an object in the transient heap. The middleware object might be a live object.
2	RESETTRACEEVENT_COMPACTED_TH The nonsystem heap containing the transient and middleware heaps has been compacted.

Reset trace events

Appendix C. Application Programming Interface

This appendix describes the application programming interface (API) for the Persistent Reusable JVM. The API comprises the following hooks, JNI functions, Java™ classes, interfaces, and methods:

abort hook

Syntax

```
void (JNICALL *abort)(void);
```

Description
A JVM abort hook.

Invocation
This function is called by the JVM during abnormal termination.

Parameters
None.

Returns
Nothing.

Example

```
void JNICALL abort_hook ()
{
    /* Tidy up */
}
```

exit hook

Syntax

```
void (JNICALL *exit)(jint code);
```

Description
A JVM exit hook.

Invocation
This function is called by the JVM during normal termination.

Parameters

code Exit code.

Returns
Nothing.

If the JNI function `DestroyJavaVM()` has been called, the "code" parameter is 0. If the `System.exit()`, `Runtime.exit()`, or `Runtime.halt()` method has been called, the "code" parameter contains the value passed on the method.

Example

```
void JNICALL exit_hook (jint code)
{
    /* Tidy up */
}
```

vfprintf hook

Syntax

```
jint (JNICALL *vfprintf)(FILE *fp, const char *format, va_list args);
```

Description

A hook function that redirects all VM messages.

Invocation

This function is called by the JVM when directing messages to stderr or stdout.

Parameters

fp File pointer to stream.

format Format string.

args Data to be written to the stream.

Returns

The number of characters printed to the stream.

Example

```
FILE *log;

jint JNICALL vfprintf_hook (FILE *fp, const char *format, va_list args)
{
    /* Log data to previously opened file */
    vfprintf(log,format,args);
}
```

JNI_a2e_vsprintf helper function

For a description of this function, see http://www-1.ibm.com/servers/eserver/zseries/software/java/usingjni.html#jnia2evsprintf.

com.ibm.jvm.classloader.Middleware interface

Syntax

```
public interface Middleware
```

Description

This tagging interface allows a class loader supplied by a middleware vendor to be tagged as a middleware class loader. There are no methods.

com.ibm.jvm.classloader.Shareable interface

Syntax

```
public interface Shareable
```

Description

This tagging interface allows a class loader supplied by a middleware vendor to be tagged as a shareable class loader. Classes loaded by this class loader are treated as shareable, and are loaded into the system heap if the class loader is also tagged with the Middleware interface, otherwise they are loaded into the application-class system heap. There are no methods.

com.ibm.jvm.ExtendedSystem class

Syntax

```
com.ibm.jvm.ExtendedSystem
```

Description

This class contains static methods and flags for the Persistent Reusable JVM.

Note: Because this class is specific to the Persistent Reusable JVM, any Java code using this class will not build with another JVM.

Member summary

```
Public Static Methods
    isJVMUnresettable()
    registerShareableClassLoader()
```

isJVMUnresettable method

Syntax

```
public static native boolean isJVMUnresettable();
```

Description

This method returns the current status of the JVM unresettable flag. It is valid to call this method only if the JVM was started with the **-Xresettable** option.

Invocation

This method is called by standard JVM class-method invocation.

Parameters

This method takes no parameters.

Returns

Returns boolean true if the JVM is unresettable; false otherwise.

isResettableJVM method

Syntax

```
public static native boolean isResettableJVM();
```

Description

This method returns true if the JVM was created with the **-Xresettable** option. It will always return false on platforms that do not support resettable mode.

Invocation

This method is called by standard JVM class-method invocation.

Parameters

This method takes no parameters.

Returns

Returns boolean true if the JVM was created with the **-Xresettable** option; false otherwise.

registerShareableClassLoader method

Syntax

```
public static void registerShareableClassLoader (
    Shareable s,
    String namespace) throws NamespaceException ;
```

Description

This method enables a class loader supplied by a middleware vendor to be registered as a shareable class loader with the Persistent Reusable JVM.

Note: It is necessary to create and register a new shareable application class loader after each JVM-reset of a Persistent Reusable JVM.

Invocation

This method must be called either from the constructor of a shareable class loader, or explicitly by middleware after creating the shareable class loader (but before using it).

Arguments

s
An instance of a class loader that implements the Shareable interface.

namespace
The string name “namespace” to be associated with the class loader.

Returns

There is no return value.

com.ibm.jvm.NamespaceException class

Syntax

`public class NamespaceException extends Exception`

Description

This is the parent exception for all namespace-related exceptions thrown by `registerShareableClassLoader()`.

Member summary

```
Public Constructors
   NamespaceException()
   NamespaceException(String s)
```

com.ibm.jvm.NamespaceInUseException class

Syntax

`public class NamespaceInUseException extends NamespaceException`

Description

This exception is thrown if a namespace has been used for another shareable class loader instance in a particular JVM.

Member summary

```
Public Constructors
   NamespaceInUseException()
   NamespaceInUseException(String s)
```

com.ibm.jvm.ClassLoaderAlreadyRegisteredException class

Syntax

`public class ClassLoaderAlreadyRegisteredException extends NamespaceException`

Description

This exception is thrown if a shareable class loader instance is already registered in a particular JVM.

Member summary

```
Public Constructors
   ClassLoaderAlreadyRegisteredException()
   ClassLoaderAlreadyRegisteredException(String s)
```

com.ibm.jvm.ClassLoaderParentMismatchException class

Syntax

`public class ClassLoaderParentMismatchException extends NamespaceException`

Description

This exception is thrown if the parent of a class loader being registered to an existing namespace does not match the parent of the previously registered class loader to the namespace.

Member summary

```
Public Constructors
   ClassLoaderParentMismatchException()
   ClassLoaderParentMismatchException(String s)
```

com.ibm.jvm.InvalidClassLoaderParentException class

Syntax

```
public class InvalidClassLoaderParentException extends
RunTimeException
```

Description

This exception is thrown if a shareable class loader is added to the hierarchy with an invalid parent. For example, a middleware class loader must also have a middleware parent.

Member summary

```
Public Constructors
   InvalidClassLoaderParentException()
   InvalidClassLoaderParentException(String s)
```

com.ibm.jvm.InvalidMiddlewareClassLoaderException class

Syntax

```
public class InvalidMiddlewareClassLoaderException extends
RunTimeException
```

Description

This exception is thrown if a middleware class is not created as a middleware object.

Member summary

```
Public Constructors
   InvalidMiddlewareClassLoaderException()
   InvalidMiddlewareClassLoaderException(String s)
```

ibmJVMReinitialize method

Syntax

```
private static void ibmJVMReinitialize();
```

Description

`ibmJVMReinitialize` methods are called the first time the class (for which the method is defined) is rereferenced after a `ResetJavaVM()`. The first time a class is referenced the static initializer is run, if present. Subsequent "first" references to the class after `ResetJavaVM()` result in this method being invoked.

The `ibmJVMReinitialize method` can be defined for any middleware class.

Invocation

This method is called by standard JVM class-method invocation.

Arguments/Parameters

This method takes no arguments or parameters.

Returns

There is no return value.

These methods are also static methods and could be used to perform some of the function of a static initializer, which is normally run by the JVM only at the first active use of the class. Indeed, a middleware class could perform one-time initialization in a static initializer which then calls `ibmJVMReinitialize` directly for the remainder. There is no return value.

Like static initializers, the order in which the `ibmJVMReinitialize` methods run is unknown and depends on when they are first rereferenced after a JVM-reset.

If an uncaught exception in a reinitialize method occurs, it is caught by the JVM. A new error com.ibm.jvm.ExceptionInReinitializerError is thrown nesting the original exception and the JVM is marked as unresettable. If the error is not handled, control returns to the launcher which is able to determine the cause of the error.

The new error is defined as follows:

```
package com.ibm.jvm;

public class ExceptionInReinitializerError extends LinkageError
{
public ExceptionInReinitializerError (Throwable thrown)
{
super(thrown);
}

:
:
}
```

ibmJVMTidyUp method

Syntax

```
private static boolean ibmJVMTidyUp();
```

Description

`ibmJVMTidyUp` methods are called for Persistent Reusable JVMs during the `ResetJavaVM()` phase to perform any necessary tidy-up of the middleware class for which they have been defined. The goal of the tidy-up is to reset the state of the middleware class to a known, predetermined value.

The `ibmJVMTidyUp` method can be defined for any middleware class.

Invocation

This method is called by standard JVM class-method invocation.

Arguments/Parameters

This method takes no arguments or parameters.

Returns

Returns TRUE if tidy-up was successful; else returns FALSE (which results in the JVM being marked as unresettable).

The order in which the `ibmJVMTidyUp` methods are called cannot be guaranteed; neither can it be configured by the middleware itself. If the order is critical to the middleware, it must supply a single `ibmJVMTidyUp` for one class which calls methods in other classes to do their tidy-up. Tidy-Up methods called in this way run serially.

The JVM calls `ibmJVMTidyUp` for all middleware classes that have been referenced since the initial creation of the JVM or the last call to JVM-reset, regardless of whether the JVM is already marked unresettable.

Note: These are static class methods, that is, they are called for each trusted middleware class, not object. If a trusted middleware class needs to perform tidy-up operations for each of its instances, it must keep track of them and deal with them individually.

If an uncaught exception in a Tidy-Up method occurs it is caught by the JVM. However, the class is not marked for reinitialization and the JVM is marked unresettable.

QueryGCStatus() JNI function

Name

`QueryGCStatus`

Syntax

```
jint QueryGCStatus(
    JavaVM *vm,
    jint *nHeaps,
    GCStatus *status
    jint statusSize);
```

Description

This JNI function can be used to report back the state of storage. The QueryGCStatus structure is defined as follows:

```
typedef struct GCStatus {
    jint  heap; // the heap that the following data is for
    jint    count; // the total number of allocation failures on this heap,
                   // or, if Querying shared storage, the subpool identifier.
                   // A negative value indicates the information is for the
                   // Shared memory Page Pool.
    jlong   freeStorage; // the total number of free bytes in the
                         // heap/subpool/page pool
    jlong   totalStorage; // the total number of bytes in the
                          // heap/subpool/page pool
} GCStatus;
```

Valid return values for heap are:

JNI_GCQUERY_TRANSIENT_HEAP

JNI_GCQUERY_MIDDLEWARE_HEAP

JNI_GCQUERY_NURSERY_HEAP

JNI_GCQUERY_MATURE_HEAP

JNI_GCQUERY_SHARED_MEM

Invocation

`QueryGCStatus` is a new JNI function and is called using the JNI invoke interface.

Arguments

vm	A pointer to a JavaVM struct
nHeaps	A pointer to the number of heaps for which data has been returned. If JNI_EINVAL is returned, it contains the number of heaps for which data is returned; this allows a user to calculate the size of status array needed to contain the data.
status	An array of GCStatus structures as defined under **Description**. This must be a pointer to an area large enough to contain data for the maximum number of heaps.
statusSize	The size of the status array in bytes.

Returns

JNI_OK	Call completed successfully
JNI_ERR	Processing the request failed
JNI_EINVAL	The query given was invalid, for example, the size of the status array is too small.

Usage Initially, the API should be called with a statusSize of 0. This call will return JNI_EINVAL and nHeaps, which is the number of storage structures for which data will be returned. This allows the caller to calculate how large statusSize should be by using something similar to nHeaps * sizeof(GCStatus).

The API returns the data in the supplied array and can be interrogated using the heap field to identify whether the data is for a heap or shared memory. If the data is for shared memory, the count field can be used to identify whether the data is for the shared memory page pool or for a subpool. There is no implied order in which the data is returned.

QueryJavaVM() JNI function

Syntax

```
jint QueryJavaVM(
    JavaVM *vm,
    jint nQueries,
    JavaVMQuery *queries);
```

Description

This JNI function can be used to query the state of the JVM at any time. The JavaVMQuery structure is defined as follows:

```
typedef struct JavaVMQuery {
    jint  name; // A valid JNI_JVMQUERY_* name
    jvalue value; // The result of the query can be anything
} JavaVMQuery;
```

Queries supported are:

JNI_JVMQUERY_EXTRA_THREADS
: Sets the corresponding value field to a jboolean of JNI_TRUE if there are additional Java threads in the JVM other than the main thread and the system threads (such as garbage collection and finalizer). Otherwise, the value is set to JNI_FALSE.

JNI_JVMQUERY_APP_FINALIZERS
: Sets the corresponding value field to a jboolean of JNI_TRUE if there are outstanding finalizers to run for application instances, that is, those in the transient heap; else the value is JNI_FALSE. For a nonresettable JVM, the value field is set to JNI_FALSE as application instances do not exist.

Invocation

| `QueryJavaVM` is a JNI function and is called using the JNI invoke interface.

Arguments/Parameters

vm	A pointer to a JavaVM struct
nQueries	The number of values to be returned
queries	An array of JavaVMQuery structures as defined under **Description**

where:

each name field is set to a valid `JNI_JVMQUERY_*` value.

Returns

JNI_OK	Call completed successfully
JNI_ERR	Processing the request failed
JNI_EINVAL	The query given was invalid

ResetJavaVM() JNI function

Syntax

```
jint ResetJavaVM(JavaVM  *vm);
```

Description

This function is called to reset a JVM that has the resettable attribute. The JVM-reset function performs the following actions:

- Allows middleware to reset itself by calling any Tidy-Up methods defined for classes that have been used
- Confirms that the JVM is resettable
- Removes application class loaders
- Performs `ResetGC` (a rapid garbage collection of the transient heap)
- Reinstates application class loaders

As soon as the JVM is found to be unresettable, a flag is set. This flag can be queried using the following method in the **com.ibm.jvm.ExtendedSystem** class:

```
public static native boolean isJVMUnresettable()
```

This method returns true if the JVM is unresettable; false otherwise.

Invocation

`ResetJavaVM` is a JNI function and is called using the JNI invoke interface.

Arguments/Parameters

vm A pointer to a JavaVM struct.

Returns

JNI_OK	Reset was successful
JNI_ERR	Reset failed
JNI_EINVAL	The JVM was not started with the **-Xresettable** option.

ResetJavaVM() JNI function

Appendix D. Notices

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

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

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

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 publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

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

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

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 IBM United Kingdom Laboratories, MP146, Hursley Park, Winchester, Hampshire, England, SO21 2JN. 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 information 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.

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.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

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

CICS	MQSeries
DB2	OS/390
IBM	

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Glossary

application-class system heap. A separate system heap for shareable application classes. Classes in this heap are loaded once only. Their static initializers are run on first active use. At JVM-reset, these classes are reset so that their static initializers rerun when next reused by a subsequent transaction application. These classes are therefore serially reusable in a resettable JVM. The initial heap size can be set by the **-Xinitacsh** parameter, and defaults to 128 KB if this is not specified. The heap expands as necessary (like the system heap).

checked standard extensions. Extensions that have been checked for writable statics and are known to be clean. Specifically, the writable static variables are either not exposed to application code, or checks have been added to the standard extension to detect any changes that are made. Loading a checked standard extension does not cause the JVM to become unresettable. However, being checked does not preclude a standard extension from making the JVM unresettable at runtime because it has performed an unresettable action. Checked extensions are identified to a Persistent Reusable JVM by being loaded from a directory that has been registered with the JVM as a checked directory.

dereference. To remove all references to a Java object by doing one of the following:

- Changing the references to another Java object
- Setting the references to NULL

It is recommended to set references to NULL.

first active use. Class initialization consists of executing the initialization code for the static fields declared in the class and the initializers for the class overall. The JVM performs initialization of the first active use of the class. A reference to a class is considered an "active use" if one of the following occurs:

- A method declared in the class (rather than a superclass) is invoked
- The constructor for the class is invoked
- A nonconstant field declared in the class (as opposed to a superclass or superinterface) is used or assigned

(Source: Sun Security Reference Model for JDK).

interned strings. In Java, all string objects are immutable and can therefore be shared. An interned string is a string from the pool of unique strings, that is, the canonical representation of the string in Java.

JNI. Java Native Interface.

| **JVMSet.** A master JVM and a set of (1-n) sharing worker JVMs.

KB. 1024 bytes.

launcher subsystem. The code that is executed to launch and control a Persistent Reusable JVM for transaction processing. Control of the Persistent Reusable JVM must be done at the JNI level so that calls to `ResetJavaVM()` can be made.

MB. 1 048 576 bytes (1024 x 1024 bytes).

middleware heap. A heap containing objects that have a life expectancy longer than a single transaction and that persist across JVM-resets. These include nonshareable class objects loaded by a user-supplied middleware class loader, and objects created in middleware context. The middleware heap forms part of the nonsystem heap. The initial size of the middleware heap can be controlled by the **-Xms** parameter.

nonsystem heap. A contiguous area of memory containing both the middleware and transient heaps. The middleware heap starts from the low end of the region and grows up, and the transient heap starts from the high end and grows down. The size of the nonsystem heap is controlled by the **-Xmx** parameter.

| **Persistent Reusable JVM.** A JVM that can be serially reused for transaction processing. This is accomplished by resetting the JVM between transactions using a new JNI function, `ResetJavaVM()`. The reset phase restores the JVM to a known initialization state. JVM reusability is enabled by the **-Xresettable** option. A Persistent Reusable JVM can be a standalone JVM or a worker JVM in a JVMSet. See *unresettable JVM*.

primordial class loader. A generic term for the bootstrap class loader and the extensions class loader.

reinitialize. For trusted middleware, the `ibmJVMReinitialize()` methods that are (optionally) defined for trusted middleware classes. These methods are called at the first active use of middleware classes after a JVM-reset. They act in the same way as a static initializer, allowing one-time actions to be performed prior to the class being used.

SAC. Shareable application class loader.

serially reusable classes. Classes that span reset cycles of the JVM, and can therefore be reused by serially executing applications running in the JVM. These classes do not need to be reloaded and re-JIT-compiled after each JVM reset.

shareable classes. Classes that are loaded by a shareable class loader. Shareable classes are serially reusable in a Persistent Reusable JVM, that is, they do not need to be reloaded by application code for each new transaction. All trusted middleware classes loaded with the Persistent Reusable JVM-supplied TMC are loaded as shareable classes. All application classes loaded with the Persistent Reusable JVM-supplied SAC are loaded as shareable classes.

system heap. The Java heap containing objects which have a life-expectancy of the life of the JVM. The objects in this heap are class objects for system classes, shareable middleware classes, and application classes. The objects are never garbage collected. The initial size of the system heap is controlled by the **-Xinitsh** option. If this is not specified, the default size is platform dependent. For OS/390, it is 128 KB. There is no maximum value restriction on the size of this heap, which expands as necessary.

tidy-up. For trusted middleware, the tidy-up phase that is performed during a JVM-reset. Tidy-up is performed by `ibmJVMTidyUp()` methods that are (optionally) defined for trusted middleware classes. Tidy-Up methods release and reset resources acquired during a transaction, and NULL out references to application objects and application class loader objects. These actions are necessary to ensure a successful JVM-reset.

TMC. Trusted middleware class loader.

transaction. A unit of application data processing (consisting of one or more application programs) initiated by a single request, often from a terminal. In Java-based transaction processing, the Persistent Reusable JVM is used to execute the Java application code associated with a transaction.

transient heap. The Java heap that contains objects with no expected lifetime beyond the end of a transaction. The objects in this heap include:

- Application objects
- Nonshareable application class objects
- Primordial objects created by application methods
- Arrays created by application methods

The transient heap is subject to normal garbage collection during the transaction, and at JVM-reset the heap is reset. The transient heap forms part of the nonsystem heap. The initial size of the transient heap can be controlled by the **-Xinitth** parameter.

trusted middleware. Java classes that can be serially reused in a Persistent Reusable JVM environment. Each middleware class is responsible for its own reset and reinitialization behavior, and can (optionally) provide Tidy-Up and Reinitialization methods to perform these operations. As a result, trusted middleware classes are not prohibited from executing Java function that is prohibited for application code. Also, middleware objects can persist across a JVM-reset. Trusted middleware classes are loaded either by a Persistent Reusable JVM-supplied class loader, which loads the classes as shareable, or by a user-defined class loader tagged with the middleware interface.

unresettable JVM. A JVM that cannot be reset to a known initialization state because either an unresettable action was performed by an application or an unresettable condition was detected. Unresettable actions are the result of calling specific Java methods that change the state of the JVM, for example, a change to a JVM static variable or system property. If such changes are made by application code, they cannot be undone when the JVM is reset because this would break Java compliance. Unresettable conditions are unrecoverable conditions that have left the JVM in either an undefined state or an error state.

writable statics. Static variables that can be accessed using public or protected methods. If an application changes the value of a writable static, the action makes the JVM unresettable. This is because the Persistent Reusable JVM cannot rely on the application resetting the static value to its initial state, and the Persistent Reusable JVM itself is prohibited from resetting static storage as this breaks Java compliance. Middleware classes can change static values because they are trusted to reset these values during the middleware tidy-up phase during JVM-reset.

Bibliography

- *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*, Li Gong, Addison-Wesley Pub Co., ISBN 0201310007.

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

W

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC34-6034-01