Oracle® JRockit JVM

Developing Java Applications

R27.6

June 2008

ORACLE

Oracle JRockit JVM Developing Java Applications, R27.6

Contents

Read the Relevant Specifications 2
Example 1: Reflection 2
Example 2: Reflection Revisited 3
Example 3: Serialization 3
Example 4: Finalizers 3
Never Use Deprecated Unsafe Methods 4
Minimize the Use of Finalizers 4
Don’t Depend on Thread Priorities 4
Don’t Use Internal sun.* or jrockit.* Classes 4
Override java.Object.hashCode for User Defined Classes When Using Hashing 5
Do Careful Thread Synchronization 5
Expect Only Standard System Properties 5
Minimizing the Number of Java Processes 6
Avoid Calling System.gc() 6
Allocate Objects Carefully 7
Be Careful When Using Signals in Native Code (JNI) 7
Use Signal Chaining 7
Don't use SIGUSR1 and SIGUSR2 7
Be Prepared to Receive Signals (Check EINTR) 8
Do You Really Need To Specify -Xrs? 8

Developing Java Applications

This document contains guidelines for writing Java applications to run on the Oracle JRockit JVM. The information provided here is in no way complete; it merely helps you avoid some common pitfalls. Oracle does not want to compromise Java's "write once run everywhere" notion. On the contrary, this document highlights guidelines that are valid for any Java program. They are, however, especially important when switching between JVMs in general and between Sun Microsystem's HotSpot JVM and the JRockit JVM in particular.

The best coding practices are summarized in the following subjects:

- Read the Relevant Specifications
- Never Use Deprecated Unsafe Methods
- Minimize the Use of Finalizers
- Don't Depend on Thread Priorities
- Don't Use Internal sun.* or jrockit.* Classes
- Override java.Object.hashCode for User Defined Classes When Using Hashing
- Do Careful Thread Synchronization
- Expect Only Standard System Properties
- Minimizing the Number of Java Processes
- Avoid Calling System.gc()

- Be Careful When Using Signals in Native Code (JNI)

Read the Relevant Specifications

Read the Java language specification and Java API specification carefully and do not rely on unspecified behavior.

The Oracle JRockit JDK is based on a number of specifications; for example, The Java Virtual Machine Specification and the Java API Specification. You should be aware that many implementations of these specifications exist: JRockit JDK is one. You should never expect any particular behavior that is *not* specified in one of these documents. Unspecified behavior might differ between the Sun JDK and the JRockit JDK. Note too that behavior is sometimes different between individual releases of the Sun JDK and can also change between releases of the JRockit JDK.

You can find these specifications at the following sites:

- The Java Language Specification

  `http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html`

- Java SE Platform API Specifications:
  - http://java.sun.com/javase/6/docs/api/index.html
  - http://java.sun.com/j2se/1.5.0/docs/api/index.html
  - http://java.sun.com/j2se/1.4.2/docs/api/index.html

- The Java Virtual Machine Specification

  `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`

The specifications are written to give JDK vendors freedom to optimize their JDKs, and therefore they leave certain behavior unspecified. You should understand, however, that numerous parts of the specifications mentioned above are unspecified. The following examples describe four of these unspecified elements.

Example 1: Reflection

The Java API Specification of the method `getMethods()` on the `java.lang.Class` class clearly states: “The elements in the array returned are not sorted and are not in any particular order.”

Example 2: Reflection Revisited

The `toString()` method of the `java.lang.reflect.Method` might include the access modifier `native`. Therefore, you should not rely on the result of this call to be equal between JVM implementations. Some classes in the Java API specification are implemented as `native` either by JRockit JDK and the Sun JDK. There is no guarantee that a native implementation on one JVM has to be native on another one.

Example 3: Serialization

The Java API Specification of the method `defaultReadObject()` of the `java.lang.ObjectInputStream` class does not specify the order in which fields are de-serialized; hence no such order can be expected.

Example 4: Finalizers

The JVM specification states that classes overriding the finalize method are guaranteed to have their finalize method run before they are garbage collected. It is up to each JVM implementation to decide when to run the finalizer method. If the object is never garbage collected, then the finalizer method is never run. Applications can thus see different kinds of starvation problems if they rely on finalizers to free certain resources such as sockets or other file handles.

In the JRockit JVM, a separate thread takes care of running the finalizer method. There are several consequences to the fact that finalizers are run in a separate thread:

- The finalizer is called by a different thread than the one that created the object.
- Only a single finalizer is processed at a time. If the finalizer blocks, or takes long time to complete, other finalizers are delayed before they are invoked.

If the application calls `System.runFinalization()`, then a secondary finalizer thread is created. This helps the primary finalizer thread to invoke unprocessed finalizers, which can shorten the time until a finalizer is run after it has been determined to be unreachable by the garbage collector. When there are no more pending finalizers available, the secondary finalizer thread finishes. This secondary finalizer thread is started with normal priority.

Creating a secondary finalizer thread can degrade performance if the finalizers are competing for a common lock.

Never Use Deprecated Unsafe Methods

Many deprecated methods are inherently unsafe and should never be used. You can see which methods are using deprecated methods by using the `-deprecation` option during compilation. For more information see:

http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html

Minimize the Use of Finalizers

Finalizers are often error prone since they often implicitly depend on the order of execution. This order differs amongst JVMs and between consecutive runs on the same JVM. Using finalizers is also inherently bad for performance since it imposes an additional burden on the memory management system that needs to handle execution of finalizers and let objects live longer.

For more information on using—and not using—finalizers, please refer to:

http://access1.sun.com/techarticles/weak.references.html

http://www.memorymanagement.org/glossary/r.html#reference.object

Don't Depend on Thread Priorities

Be careful when using `java.lang.Thread.setPriority`. Depending on thread priorities might lead to unwanted or unexpected results since the scheduling algorithm might choose to starve lower priority threads of CPU time and never execute them. Furthermore the result might differ between operating systems and JVMs.

The Java API specification states that "Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority."

The priority set by the `setPriority()` method is a parameter that might be used in the thread-scheduling algorithm, which shares CPU execution time between executing threads. This algorithm might be controlled either by the JVM or by the operating system. It is important to be aware of the fact that this algorithm normally differs between operating systems and that the algorithm might change between releases of both the operating system and the JVM.

Don't Use Internal sun.* or jrockit.* Classes

The classes that the Oracle JRockit JDK includes fall into package groups `bea.*`, `java.*`, `javax.*`, `org.*`, `sun.*`, `com.bea.jvm.*` and `jrockit.*`. All except the `sun.*`, `jrockit.*` and `com.bea.jvm.*` packages are standard to the Java platform and will be supported into the

future. The `com.bea.jvm.*` classes contain both documented and officially supported classes, along with internal, unsupported classes. This paragraph refers to the non-documented `com.bea.jvm.*` classes. Generally, non-standard packages, which are outside of the Java platform, can be different across JVM vendors and OS platforms (Windows, Linux, and so on) and can change at any time, without notice, with JDK versions. Programs that directly use the `sun.*` and `jrockit.*` packages are not 100% Pure Java.

For more information, please refer to the note about `sun.*` packages at:

`http://java.sun.com/products/jdk/faq/faq-sun-packages.html`

Override java.Object.hashCode for User Defined Classes When Using Hashing

In the JRockit JVM, the current default implementation of `hashCode` returns a value for the object determined by the JVM. The value is created using the memory address of the object. However, because this value can be reused if the object is moved during garbage collection, it is possible to obtain the same hash code for two different objects. Also, two objects that represent the same value are guaranteed to have the same hash code only if they are the exact same object. This implementation is not particularly useful for hashing; therefore, when objects of the classes should be stored in a `java.util.Hashtable` or `java.util.HashMap`, derived classes should override `hashCode()`.

Do Careful Thread Synchronization

Make sure that you synchronize threads that access shared data. Synchronization bugs often appear when changing JVMs because the implementation of locks, garbage collection, thread scheduling and so on, might differ significantly.

Note: If your code contains synchronization problems (deadlocks and race conditions), please note that the bugs already existed, even on the other JVM. If similar synchronization problems did not arise when you used the other JVM, it was by pure chance.

Expect Only Standard System Properties

When calling `java.lang.System.getProperties()` or `java.lang.System.getProperty()`, you should only depend on standard system properties being returned; different VMs may return a different set of extended properties. Non-standard properties should not be returned.

When the JVM starts, it inserts a number of standard properties into the system properties list. These properties, and the meaning of their values, are listed in the Java API specification. Do not rely on any other non-standard properties.

http://java.sun.com/j2se/1.5.0/docs/api/java/lang/System.html#getProperties()

Minimizing the Number of Java Processes

When designing applications there is sometimes a choice between running several processes, i.e. JVM instances versus running several threads or thread groups within a single process, i.e. JVM instance. In most cases, it is more effective to use as few JVM instances as possible per physical machine.

Avoid Calling System.gc()

Do not call `java.lang.System.gc()`. The JRockit JVM garbage collector generally does a much better job than `System.gc()` in deciding when to do garbage collection. In fact, performance is likely to decrease if your application repeatedly calls `System.gc()`. If you are having problems with memory usage, pause times for garbage collection, or similar issues, then you should configure the memory management system appropriately. See Garbage Collection, Memory Management, and Tuning For Faster Individual Transactions (all part of the Oracle JRockit *Diagnostics Guide*).

Bear in mind that this method can behave differently on the JRockit JVM than on other JVMs. On the JRockit JVM, calling `System.gc()` results in a nursery collection if you are using a generational collector, and a old space collection if you're using a single generational collector.

You can change the behavior of `System.gc()` by the following three command line options:

- `-XXnoSystemGC`: If this command line option is given, calls to `System.gc()`are ignored, and no garbage collection takes place as a result of these method calls. This can be beneficial if you are using third party libraries that call `System.gc()`, in which in case the performance of your application may otherwise be negatively affected.
- `-XXfullSystemGC`: If this command line option is given, calls to `System.gc()` always results in an old space collection, even if you are using a generational collector. In addition, a full collection that eliminates soft references will be performed.
- `-XXprintSystemGC`: If this command line option is given, all calls to `System.gc()` are logged. This can be helpful to determine if your application is causing bad performance by invocations of `System.gc()`.

Allocate Objects Carefully

Object allocation causes garbage collection. Try to avoid object allocation and reallocation when possible, for example:

- Avoid using ArrayList or HashMap for a dynamically growing data structure, as the entire structure is reallocated when it grows.
- Avoid creating unnecessary temporary copies of large objects or large amounts of data.

Be Careful When Using Signals in Native Code (JNI)

The JRockit JVM uses signals internally for various purposes. If you want to use a native library that employs signals, you should keep adhere to these guidelines:

- Use Signal Chaining
- Don't use SIGUSR1 and SIGUSR2
- Be Prepared to Receive Signals (Check EINTR)
- Do You Really Need To Specify -Xrs?

Use Signal Chaining

Mixing Java and signalling requires the use of signal chaining, a feature that enables Java to better inter-operate with native code that installs its own signal handlers. You can chain signals by either linking with `libjsig.so` at link time or by using `LD_PRELOAD` environment variable at runtime. These chaining techniques are described in detail in the Sun Microsystems document, Signal Chaining, at:

`http://java.sun.com/j2se/1.5.0/docs/guide/vm/signal-chaining.html`

Don't use SIGUSR1 and SIGUSR2

The JRockit JVM uses the two signals `SIGUSR1` and `SIGUSR2` for VM-internal purposes. These signals are essential to the functionality of the Oracle JRockit JVM and cannot be disabled. Thus, no user native library may use `SIGUSR1`. Any such use will cause the JVM to fail, typically by hanging the thread that received the signal (for `SIGUSR1`) or crashing the JVM (for `SIGUSR2`).

Be Prepared to Receive Signals (Check EINTR)

Native code that calls system routines (such as sleep) should *always* check if the function sets `errno` to `EINTR` if it returns failure (`-1`). If so, the system call should typically just be retried. (Some exceptions apply, check your system's programming manual.) Since the JRockit JVM sends `SIGUSR1` to attached threads at regular intervals, do not be surprised if you get signals everywhere in your code.

Do You Really Need To Specify -Xrs?

When the `-Xrs` ("reduce signals") command-line option is specified, the signal masks for SIGINT, `SIGTERM`, `SIGHUP`, and `SIGQUIT` are not changed by the JVM, and signal handlers for these signals are not installed. There are two consequences of specifying `-Xrs`:

- `SIGQUIT` thread dumps are not available.
- User code is responsible for causing shutdown hooks to run, for example by calling `System.exit()` when the JVM is to be terminated.

The option is called "reduce signals" because it reduces the use of operating-system signals by the JVM.