Oracle® JRockit

Application Development Guide

Release R28

E15063-03

November 2010

This document contains guidelines for writing Java applications to run on the Oracle JRockit JVM. These guidelines are especially important when switching between JVMs in general, and between the HotSpot JVM and the JRockit JVM in particular.

This document contains the following sections:

1 Read the Relevant Specifications

Read the Java language and Java API specifications 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 can find these specifications at the following sites:

- 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`
- Java Virtual Machine specification

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

Many implementations of these specifications exist: the JRockit JDK is one. Do not expect any particular behavior that is *not* specified in one of these documents. Unspecified behavior might differ between the HotSpot JDK and the JRockit JDK. Note that the behavior sometimes differs between individual releases of the HotSpot JDK or between releases of the JRockit JDK.

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

1.1 Example 1: Reflection

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

1.2 Example 2: Reflection Revisited

The `toString()` method of the `java.lang.reflect.Method` class 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 the JRockit JDK or the HotSpot JDK. There is no guarantee that a native implementation on one JVM will be native on another one.

1.3 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 deserialized, therefore, no such order can be expected.

1.4 Example 4: Finalizers

The JVM specification states that classes that override the `finalize` method are guaranteed to have their `finalize` method run before the objects are garbage collected. It is up to each JVM implementation to decide when to run the finalizer. If the object is never garbage collected, then the finalizer 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. There are two 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 a long time to complete, other finalizers are delayed before they are invoked.

If the application calls the `System.runFinalization()` method, 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 a normal priority.

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

2 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 about deprecated methods, see:

- Java SE 6 documentation

  http://java.sun.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html

- J2SE 5.0 documentation

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

3 Minimize the Use of Finalizers

Finalizers are often error prone because they often implicitly depend on the order for execution. This order differs among JVMs and between consecutive runs on the same JVM. Using finalizers can slow performance because it imposes an additional burden on the memory management system that must handle execution of finalizers and let objects remain active.

For more information about finalizers, see http://www.memorymanagement.org/glossary/r.html#reference.object.

4 Do Not Depend on Thread Priorities

Be careful when using the `java.lang.Thread.setPriority` method. Depending on thread priorities might lead to unwanted or unexpected results because the scheduling algorithm might not allocate low-priority threads adequate CPU time or never execute them. Furthermore, the result might differ between operating systems and JVMs.

The Java API specification states that every thread has a priority and 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 know that this algorithm usually differs between operating systems, and that the algorithm might change between releases of both the operating system and the JVM.

5 Do Not Use Internal sun.* or jrockit.* Classes

The classes that the Oracle JRockit JDK includes divide among 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 are supported. The `com.bea.jvm.*` classes contain both documented and officially supported classes, along with internal, unsupported classes.

This information refers to the undocumented `com.bea.jvm.*` classes. Generally, nonstandard packages, which are outside of the Java platform, can be different across JVM vendors and operating systems (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 entirely Java.

For more information, see the note about `sun.*` packages at:

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

6 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 as 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` class. derived classes should override the `hashCode()` method.

7 Synchronize Threads Carefully

Ensure 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), note that the bugs already existed, even on the other JVM. If similar synchronization problems did not arise when you used the other JVM, the synchronization problems were purely conincidental.

8 Expect Only Standard System Properties

When calling the `java.lang.System.getProperties()` or `java.lang.System.getProperty()` methods, expect only standard system properties to be returned; different JVMs may return a different set of extended properties. Nonstandard 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 specifications. Do not rely on any other nonstandard properties. For more information about these APIs, see:

- Java SE 6 API specification

  `http://java.sun.com/javase/6/docs/api/java/lang/System.html#getProperties()`

- J2SE 5.0 API specification

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

9 Minimize the Number of Java Processes

When designing applications, you have a choice between running several processes, that is, JVM instances or running several threads or thread groups within a single process, a JVM instance. In most cases, it is more effective to use as few JVM instances as possible per computer.

10 Avoid Calling System.gc()

Do not call the `java.lang.System.gc()` method. The JRockit JVM garbage collector determines when to do the garbage collection better than the `System.gc()` method. The performance is likely to decrease if your application repeatedly calls the `System.gc()` method. If you have problems with memory usage, pause times for garbage collection, or similar issues, then configure the memory management system appropriately. For more information about performance tuning, see the *Oracle JRockit Performance Tuning Guide*.

This method can act differently on the JRockit JVM than on other JVMs. On the JRockit JVM, calling the `System.gc()` method results in a nursery collection if you are using a generational collector, and an old space collection if you are using a single generational collector.

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

- `-XXnoSystemGC`: If this command-line option is given, calls to the `System.gc()` method 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 the `System.gc()` method, in which in case, the performance of your application may be negatively affected.
- `-XXfullSystemGC`: If this command-line option is given, calls to the `System.gc()` method always result 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.
- `-Xverbose:systemgc`: If this command-line option is given, all calls to the `System.gc()` method are logged. This can be helpful to determine if your application is causing bad performance by invocations of the `System.gc()` method.

11 Allocate Objects Carefully

Object allocation causes garbage collection. Avoid object allocation and reallocation when possible, for example:

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

12 Be Careful When Using Signals in Native Code

The JRockit JVM uses signals internally for various purposes. If you want to use a native library (Java Native Interface) that employs signals, adhere to these guidelines:

- Use Signal Chaining
- Do Not Use SIGUSR1 and SIGUSR2
- Be Prepared to Receive Signals (Check EINTR)
- Ensure That You Really Need to Specify -Xrs

12.1 Use Signal Chaining

Mixing Java and signalling requires the use of signal chaining, a feature that enables Java to better interoperate 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 the `LD_PRELOAD` environment variable at runtime. These chaining techniques are described in detail in the Java documents for Signal Chaining at the following locations:

- Java SE 6

  `http://java.sun.com/javase/6/docs/technotes/guides/vm/signal-chaining.html`

- J2SE 5.0

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

12.2 Do Not Use SIGUSR1 and SIGUSR2

The JRockit JVM uses the two signals `SIGUSR1` and `SIGUSR2` for JVM-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 the `SIGUSR1` signal. 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`).

12.3 Be Prepared to Receive Signals (Check EINTR)

Native code that calls system routines (such as `sleep`) should always check to determine if the function sets `errno` to `EINTR` and returns failure (`-1`). If so, the system call should typically be retried (some exceptions apply; check your system's programming manual). The JRockit JVM sends the `SIGUSR1` signal to attached threads at regular intervals. Therefore, you might get signals everywhere in your code.

12.4 Ensure That 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 the `System.exit()` method 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.

13 Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/support/contact.html` or visit `http://www.oracle.com/accessibility/support.html` if you are hearing impaired.

Oracle JRockit Application Development Guide, Release R28
E15063-03