

Excelsior Knowledge Base

INFO: Implementation-specific differences between JVMs

Article ID: 000032

Last Revised On: 21-Apr-2010

The information in this article applies to:

- Excelsior JET 7.0 and above (all editions)

You may download the latest version of Excelsior JET [here](#).

INTRODUCTION

Each released version of Excelsior JET passes the Java Compatibility Kit (JCK) test suite. This guarantees that all features of the Java programming language and Java SE platform are implemented in compliance with the specification. However, certain applications exploit Java features not enforced by the specification. Such applications may fail to run on the JET Runtime, and probably on some other JVMs too.

DETAILS

This section lists the known implementation differences between Excelsior JET and the HotSpot JVM, the reference implementation included with the Sun-Oracle JRE. You are encouraged to remove all possible usages of implementation-biased features to make your application compatible with any compliant Java SE VM.

VM vendor

The first obvious distinction is that the `java.vm.vendor` system property has a different value. Surprisingly, certain third-party Java components use it to determine the "capabilities" of the underlying JVM.

How to fix: You may try to find the latest versions of such components, which may have been improved to work with a larger set of Java SE VMs.

Order of reflected class members

The elements in the array returned by `Class.getDeclaredConstructors()`, `Class.getDeclaredMethods()`, and `Class.getDeclaredFields()` are not sorted and are not in any particular order.

This is exactly what the Java SE API specification says rather than a feature of Excelsior JET. Nevertheless, certain third-party Java components tries to exploit a particular order of reflected class members.

How to fix: You may try to find the latest versions of such components, which may have been improved to work with a larger set of Java SE VMs.

Deprecated methods of class Thread

`Thread.stop()` has no effect unless the thread is waiting on an object. `Thread.suspend()` may provoke a deadlock in the JET Runtime.

Note that `Thread.stop()`, `Thread.suspend()`, and `Thread.resume()` are deprecated methods. Specifically for Excelsior JET, it is *strongly* recommended to rewrite the code that uses these deadlock-prone methods.

How to fix: A detailed explanation and recommended replacements are given in the article "*Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?*" available at

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

Eager class resolution

When compiling a Java application, the JET Optimizer tries to resolve almost all references from each class to other imported classes. On the contrary, HotSpot exploits lazy class resolution, that is, a reference to a class is resolved only when it is first used. Both resolution policies are allowed by the Java specification.

Reliance on lazy class resolution in the application code may cause subtle side-effects and, in general, results in creation of fragile code. It is better to avoid using speculative assumptions on when a particular class will be loaded.

Workaround: If the code exploiting lazy class resolution is not pre-compiled, that is, it is handled by the JIT compiler included with the JET Runtime, you may enable the `jet.jit.disable.resolution` VM option that enforces lazy classloading in the JITted classes.

No JVMTI

JVM Tool Interface (JVMTI), an optional part of the Java SE platform API, is not supported in Excelsior JET due to security reasons. Its implementation would enable a wide variety of introspection techniques for the compiled applications, which is highly undesirable.

No class/jar files

To provide code protection, the original class/jar files are not deployed with the compiled application, and therefore, are not available at run time. If the application tries, for example, to compute a check sum of a class by reading its content with conventional file system operations, it would fail.

Workaround: You may configure the JET Optimizer to make the required jar file available to the compiled application, provided they do not contain the code you wish to protect from decompilation. For more details, see the JET User's Guide, chapter "Application consideration", section "Resource packing". Moreover, if the class files of the Java SE API are also required, you may enable packaging them along with application on page Runtime of JetPackII (see JET User's Guide, chapter "Deployment automation", section "Configuring the JET Runtime".)

Note: Some applications, such as the `javac` compiler, need only symbolic information from class files and do not use the contained bytecode instructions. In such cases, everything should work without any workarounds.

Standard Exception Messages

Detailed messages of the exceptions thrown by the JVM, such as `IndexOutOfBoundsException`, may differ from those of HotSpot.

Stack trace

By default, `Throwable.printStackTrace()` methods print a few fake elements, because stack tracing is disabled in the JET Runtime. However, certain third-party APIs may rely on stack trace printing. For example, the Log4J API uses the stack trace information providing logging services.

Workaround: You may enable stack trace printing as described in the JET User's Guide, chapter "Application consideration", section "Stack trace".

Default heap size

If you do not set the maximum heap size, the JET Runtime determines it adaptively depending on the allocation profile and amount of available physical memory as described in the JET User's Guide, chapter "Application consideration", section "Memory management".

The HotSpot JVM uses different heuristics to set the default maximum heap size. For more details, see <http://java.sun.com/j2se/1.5.0/docs/guide/vm/gc-ergonomics.html>

Workaround: You may set the maximum heap size explicitly.

Signed jars

As, by default, the application jar files are not deployed with the compiled executable, checking digests of the signed jars fails.

Workaround: You may configure the JET Optimizer to make the signed jars available to the compiled application as described in the JET User's Guide, chapter "Application consideration", section "Resource packing".

Note: In such case, the jars will probably be used only for checking the signatures which does not help ensure the authenticity. For that purpose, it would make more sense to sign the compiled executable using a third-party utility.

Non-standard VM options

Most `-XX` options are not supported as they are specific to a particular implementation. Moreover, certain `-X` options are not supported too, for example setting `-Xbootclasspath` takes no effect. The list of the supported VM options is given in the JET User's Guide, chapter "Application consideration", section "Standard Runtime options".

Endorsed jars

If your application uses Endorsed Standards Override, it is not enough to specify the `java.endorsed.dirs` system property as you do when using the HotSpot JVM. To enable the override mechanism, you need to use the JET Setup utility to create a new JET profile including the endorsed jars. For more details, see the JET User's Guide, chapter "Installation and setup", section "Profiles".

REFERENCES

1. Excelsior JET User's Guide.