BEA JRockit

Using BEA JRockit JDK

JRockit 5.0 Service Pack 2
June 2005

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

Introduction

Starting and Configuring BEA JRockit JVM

Using the BEA JRockit Memory Management System

Using the Java Plug-in

Using Web Start with BEA JRockit

Using jstat with BEA JRockit

Monitoring Thread Activity With Thread Dumps

Index

CHAPTER 1

Introduction

Welcome to Using BEA JRockit JDK. This document contains procedures and other information necessary for you to gain optimal performance from BEA Systems' industry-leading Java Virtual Machine, BEA JRockit.

This Introduction includes information on the following subjects:

- What's In the User Guide?
- Finding Additional Information

What's In the User Guide?

This user guide is organized as follows:

- Starting and Configuring BEA JRockit JVM describes how to start and configure your JRockit for the best performance for your application. You also find information on how to use the extended options in JRockit.
- Using the BEA JRockit Memory Management System describes how to use the best memory management system—or garbage collection method—for your application.
- Using the Java Plug-in describes how to install and run the Java Plug-in for your web browser.
- Using Web Start with BEA JRockit describes how to install and use Web Start on your system.

- Monitoring Thread Activity With Thread Dumps describes how you use your stack dumps to follow the thread activity in JRockit.

Finding Additional Information

You can find additional information about BEA JRockit throughout the documentation set. For a complete list of available documents, please refer to BEA JRockit JDK Online Documentation. The following list cites the most commonly referenced information.

BEA JRockit Support

To get support for BEA JRockit, you need a service agreement with BEA. If you want to be part of the JRockit discussion group, please go to the BEA JRockit news group: http://forums.bea.com/bea/category.jspa?categoryID=2010

Supported Platforms

For a list of platforms supported by BEA JRockit, please refer to "Supported Platforms".

Tuning BEA JRockit

Tuning information can be found in Tuning BEA JRockit JVM.

CHAPTER 2

Starting and Configuring BEA JRockit JVM

This section describes how to start BEA JRockit and how to configure it by using standard and non-standard command line options. It includes information on the following subjects:

- Before Starting BEA JRockit
- Starting BEA JRockit
- Configuring BEA JRockit

Before Starting BEA JRockit

Before starting BEA JRockit, ensure that you have the following directory set in your PATH environment variable:

- `<jrockit-install-directory>/bin` (for Linux)
- `<jrockit-install-directory>\bin` (for Windows)

Starting BEA JRockit

To start BEA JRockit, at the command line enter the following:

`java <configuration and tuning options> myClass`

Where `<configuration and tuning options>` are the optional configuration and tuning options you want to use. The configuration options are described in Configuring BEA JRockit, below. See Tuning BEA JRockit JVM for details on the tuning options available for this version of BEA JRockit.

Note: You can alternatively start JRockit by specifying the full path to the file; for example, `/usr/local/java/bin/java` (depending on where it is installed) on Linux and `c:\bea\jrockit`*xxx*`\bin\java` (depending on where it is installed) on Windows.

Setting the JRockit Type

The following commands set the type of JRockit you want to run, server-side or client-side:

- `-server`

  Starts BEA JRockit as a server-side JVM. This option is default.

- `-client`

  Starts BEA JRockit as a client-side JVM. This option is helpful if you have a smaller heap and are anticipating shorter runtimes for your application.

By setting the JVM type (or accepting the default) will also set the garbage collection algorithm that will be used during runtime. `-server` will start the dynamic garbage collector optimized for throughput while `-client` will start a a single-spaced, concurrent mark, concurrent sweep garbage collector. If you want to use a specific fixed garbage collector, you can override the default by using the `-Xgc` command line option.

Sample Start-up Command

A sample start-up command, with some tuning options specified, might look like this:

```
java -Xverbose:memory -Xmx:256m -Xms:64m myClass
```

In this example, the following options are set:

- `-Xverbose:memory`—Displays verbose output about memory usage.
- `-Xmx:256m`—The maximum heap size is set to 256 megabytes.
- `-Xms:64m`—The initial and minimum heap size is set to 64 megabytes.
- `myClass`—Identifies the class that contains the `main` method.

Configuring BEA JRockit

When you start BEA JRockit, you can set behavioral parameters by using both standard and non-standard command line options. This section describes some of these options and how to use them at startup to configure BEA JRockit. It contains information on the following subjects:

Using Standard Command Line Options

The standard command line options work the same regardless of the JVM; in other words, these options work the same whether you are running BEA JRockit, Sun Microsystem's HotSpot JVM, or any other third party JVM.

Setting General Command Line Options

The following standard command line options set general information about BEA JRockit:

- `-classpath <directories and zips/jars separated by : (Linux) or ; (Windows)>`

  Specifies the location of classes and resources.

  Alternately, you can use the option `-cp` to represent `-classpath`; for example:

  ```
  -cp <directories and zips/jars separated by : or ;>
  ```

- `-D<name>[=<value>]`

  Specifies a Java system property. These can be read by a Java program, using the methods in `java.lang.System`.

Providing Information to the User

The following options determine if the system will provide messages to the operator and what the form and content of those messages should be.

- `-help`

  Displays a short help message.

- `-version`

Displays the product version of JRockit and then exits.

- `-showversion`

  Displays the product version of JRockit and then continues with the operation.

- `-verbose`

  Displays verbose output. This option is used mainly for debugging purposes and causes a lot of output to the console.

Using Extended Options

Extended command line options, proceeded with the letter `-X`, are options that are exclusive to BEA JRockit and changes the behavior of JRockit to better suit the needs of different Java applications. These options will not work on other JVMs (conversely, the extended options used by other JVMs won't work with JRockit).

The option `-X` displays a short help message on the extended options.

Note: Since these options are an extension to JRockit and non-standard, they are subject to change between releases, see the BEA JRockit JDK Compatibility Statement.

Setting Behavioral Options

The following are examples on extended options that define general BEA JRockit JVM behavior:

- `-Xns`

  Sets the size of the nursery in a generational garbage collector.

- `-Xms`

  Sets the initial size of the heap.

- `-Xgc`

  Sets a specific fixed garbage collector.

- `-Xgcprio`

  Sets the dynamic garbage collector.

Providing Information to the User

When using the startup option `-Xverbose`, BEA JRockit prints, on screen, specific information about the system. The information printed depends upon the parameter that you have specified

with the option. Supported parameters are, for example, `memory`, `load`, `gc`, `opt`, and `cpuinfo`. If you do not specify any parameter, everything will be printed.

Note: To use more than one parameter, separate them with a comma, for example: `-Xverbose:gc,opt`

Listing 2-1 through Listing 2-6 combined with Table 2-1 through Table 2-6 lists and explains examples of different verbose output. These output examples are to hint you to how the verbose output can look like. The output that you see in these listings can greatly differ from what you see on your system depending on, for example, the version of BEA JRockit that you are running.

Listing 2-1 Print out for -Xverbose:codegen

```
[codegen] #1 ? (0x2) n
jrockit/vm/Allocator.prepareNextChunkAndAlloc(IIII)Ljava/lang/Object;
[codegen] #1 ? (0x2) n @0x6b3543f0-0x6b354465  1.43 ms (1.43 ms)
```

Table 2-1 Explanation of Listing 2-1

Code snippet	Explanation
`[codegen] #1 ? (0x2) n`	This is the first (#1) method to be generated with a normal (n) or non-optimized code generation. n means non-optimized code generation o means optimized code generation q means quick code generation
`jrockit/vm/Allocator.prepareNextChunkAndAlloc(IIII)Ljava/lang/Object;`	This is the name and location of the method that has been generated.
`@0x6b3543f0-0x6b354465`	The address in the memory where the method resides.
`1.43 ms`	The time it took to generate the code.
`(1.43 ms)`	The total time that JRockit has generated code.

Listing 2-2 Print out for -Xverbose:opt

```
[opt    ] #1 4 (0x8) o
jrockit/vm/Locks.waitForThinRelease(Ljava/lang/Object;I)I
[opt    ] #1 4 (0x8) o @0x324D0000-0x324D00A1  26.80 ms (26.80 ms)
```

Table 2-2 Explanation of Listing 2-2

Code snippet	Explanation
`[opt    ] #1 4 (0x8) o`	This is the first (#1) method to be generated with an optimized (o) code generator. n means non-optimized code generation o means optimized code generation q means quick code generation
`jrockit/vm/Locks.waitForThinRelease(Ljava/lang/Object;I)I`	This is the name and location of the method that has been generated.
`@0x324D0000-0x324D00A1`	The address in the memory where the method resides.
`26.80 ms`	The time it took to generate the code.
`(26.80 ms)`	The total time that JRockit has generated code.

Listing 2-3 Print out for -Xverbose:cpuinfo

```
[cpuinfo] Vendor:   GenuineIntel
[cpuinfo] Type:     Original OEM
[cpuinfo] Family:   Pentium 4
[cpuinfo] Brand:    Intel(R) Xeon(TM) CPU 2.80GHz
[cpuinfo] Supports: On-Chip FPU
[cpuinfo] Supports: Virtual Mode Extensions
[cpuinfo] Supports: Debugging Extensions
```

Table 2-3 Explanation of Listing 2-3

Code snippet	Explanation

`[cpuinfo] Vendor: GenuineIntel`
`[cpuinfo] Type: Original OEM`
`[cpuinfo] Family: Pentium 4`
`[cpuinfo] Brand: Intel(R)`
`Xeon(TM) CPU 2.80GHz`
	This is information about the CPU chip itself, i.e. vendor, type of chip, family name, and the brand name. This information differs depending on the type of CPU you are using.

`[cpuinfo] Supports: On-Chip FPU`
`[cpuinfo] Supports: Virtual Mode`
`Extensions`
`[cpuinfo] Supports: Debugging`
`Extensions`
`.`
`.`
`.`
	This lists all features that the CPU supports. For more information about your specific CPU, please contact the vendor.

Listing 2-4 Print out for -Xverbose:load

```
[load   ] opened zip
/localhome/jrockits/jrockit-jdk1.5.0_02/jre/lib/jrockit.jar
[load   ] opened zip /localhome/jrockits/jrockit-jdk1.5.0_02/jre/lib/rt.jar
[load   ] opened zip
/localhome/jrockits/jrockit-jdk1.5.0_02/jre/lib/jsse.jar
[load   ] opened zip
/localhome/jrockits/jrockit-jdk1.5.0_02/jre/lib/jce.jar
[load   ] opened zip
/localhome/jrockits/jrockit-jdk1.5.0_02/jre/lib/charsets.jar
[load   ] opened zip
/localhome/jrockits/jrockit-jdk1.5.0_02/jre/lib/managementapi.jar
[load   ] initiated ? (0x2)   0 (nil)/java/lang/Object
[load   ] define ? (0x2) #  0 java/lang/Object loader=(nil),
src=/localhome/jrockits/jrockit-jdk1.5.0_02/jre/lib/jrockit.jar
[load   ] loading ? (0x2)   0 (nil)/java/lang/Object success (0.59 ms)
```

Table 2-4 Explanation of Listing 2-4

Code snippet	Explanation
`[load ] opened zip /localhome/jrockits/jrockit-jdk1.5.0_02/jre/lib/jrockit.jar`	This is information about the classes loaded in JRockit.
`java/lang/Object`	The name of the class loaded for your application.
`src=/localhome/jrockits/jrockit-jdk1.5.0_02/jre/lib/jrockit.jar`	The address from where the class is loaded.
`success`	Shows if the class loaded successfully. If it failed, the print out is `fail`.

Listing 2-5 Print out for -Xverbose:memory (dynamic garbage collector)

```
[memory ] GC strategy: System optimized over throughput (initial strategy
singleparpar)
[memory ] heap size: 65536K, nursery size: 16384K
[memory ] <s>-<end>: GC <before>K-><after>K (<heap>K), <pause> ms
[memory ] <s/start> - start time of collection (seconds since jvm start)
[memory ] <end>     - end time of collection (seconds since jvm start)
[memory ] <before>  - memory used by objects before collection (KB)
[memory ] <after>   - memory used by objects after collection (KB)
[memory ] <heap>    - size of heap after collection (KB)
[memory ] <pause>   - total pause time during collection (milliseconds)
[memory ] Changing GC strategy to generational, parallel mark and parallel
sweep
[memory ] 1.719-1.731: GC 65536K->3176K (65536K), 11.000 ms
```

Table 2-5 Explanation of Listing 2-5

Code snippet	Explanation
`[memory ] GC strategy: System optimized over throughput (initial strategy singleparpar)`	Information about the garbage collector strategy that is used. This is the default dynamic garbage collector that optimizes over throughput.
`[memory ] heap size: 65536K, nursery size: 16384K`	This is the initial heap and nursery size.
`[memory ] <s>-<end>: GC <before>K-><after>K (<heap>K), <pause> ms`	This is the format of the verbose print. The text that follow is an explanation of the different parts of the print-out.
`[memory ] 1.719-1.731: GC 65536K->3176K (65536K), 11.000 ms`	This the first verbose print of a successful garbage collection.

Listing 2-6 Print out for -Xverbose:memory (parallel, single-spaced garbage collector)

```
[memory ] GC strategy: parallel
[memory ] heap size: 65536K
[memory ] <s>-<end>: GC <before>K-><after>K (<heap>K), <pause> ms
[memory ] <s/start> - start time of collection (seconds since jvm start)
[memory ] <end>     - end time of collection (seconds since jvm start)
[memory ] <before>  - memory used by objects before collection (KB)
[memory ] <after>   - memory used by objects after collection (KB)
[memory ] <heap>    - size of heap after collection (KB)
[memory ] <pause>   - total pause time during collection (milliseconds)
[memory ] 1.561-1.572: GC 65536K->1420K (65536K), 10.000 ms
```

Table 2-6 Explanation of Listing 2-6

Code snippet	Explanation
`[memory ] GC strategy: parallel`	Information about the garbage collector strategy that is used. Here it is a static parallel, single-spaced garbage collector.
`[memory ] heap size: 65536K`	This is the initial heap size.
`[memory ] <s>-<end>: GC <before>K-><after>K (<heap>K), <pause> ms`	This is the format of the verbose print. The text that follow is an explanation of the different parts of the print-out.
`[memory ] 1.561-1.572: GC 65536K->1420K (65536K), 10.000 ms`	This the first verbose print of a successful garbage collection.

Including a Timestamp with Logging Information

Use the startup options `-Xverbose:memory` and `-Xverbosetimestamp` or `-Xverbose:opt` and `-Xverbosetimestamp` to view a time and date stamp preceded by the other verbose information, as shown here:

```
[Thu Apr 21 10:24:11 2005][ 5656][memory ] 4.578: parallel nursery GC
22067K->7457K (65536K), 8.905 ms
```

```
[Thu Apr 21 10:24:11 2005][ 5656][memory ] 4.781: parallel nursery GC
22157K->7549K (65536K), 9.954 ms
```

Sample of timestamp information for `-Xverbose:opt` and `-Xverbosetimestamp`:

```
[Thu Apr 21 10:24:19 2005][ 5576][opt    ] #3 4 (0x8) o
jrockit/vm/Locks.monitor Enter(Ljava/lang/Object;)Ljava/lang/Object;
```

```
[Thu Apr 21 10:24:19 2005][ 5576][opt    ] #3 4 (0x8) o
@0x324D0A90-0x324D0AD4 3.29 ms (235.46 ms) 17.26 ms (252.72 ms)
```

CHAPTER 3

Using the BEA JRockit Memory Management System

Memory management relies on effective "garbage collection," the process of clearing dead objects from the heap, thus releasing that space for new objects. BEA JRockit uses a dynamic garbage collector that is based upon one of two priorities that you set: application throughput or duration of the pause times caused by garbage collection. The dynamic garbage collector uses predefined heuristics to determine, in runtime, which garbage collection algorithm to use for each application.

In some instances, dynamic garbage collection might not be the most effective way to recycle memory. In those cases, BEA JRockit also provides a number of "static" garbage collectors that can be started by specifying the actual collector (`-Xgc:<collectorName>`) at startup.

This section describes how to use all of these garbage collection methods. It contains information on the following subjects:

The Mark-and-Sweep Garbage Collection Model

The garbage collector models in JRockit are all *mark-and-sweep* garbage collectors that run either as generational or single-spaced; that is, with or without a "nursery" (see Two-generational Garbage Collection, below). The mark-and-sweep garbage collection is implemented as either a concurrent or parallel algorithm.

A mark-and-sweep garbage collector frees all unreferenced objects and works as described in these steps:

1. The "mark" phase, traverses all pointers, starting at the accessible roots of a program (conventionally, globals, the stack, and registers) and marks each object traversed.
2. The "sweep" phase, re-walks the heap linearly and removes all objects that are not marked.

Garbage Collector Options

JRockit's garbage collector can be a combination of the following two garbage collector options:

- Two-generational Garbage Collection
- Single-generational Garbage Collection

and two garbage collection algorithms:

- Concurrent Garbage Collection
- Parallel Garbage Collection

Two-generational Garbage Collection

During a two-generational garbage collection, the heap is divided into two sections: an old generation and a young generation—also called the "nursery." Objects are allocated in the nursery and when it is full, JRockit stops all Java threads and moves the live objects from the nursery, young generation, to the old generation.

Single-generational Garbage Collection

The single-spaced option of garbage collection means that all objects live out their lives in a single space on the heap, regardless of their age. In other words, a single-spaced garbage collector does not have a nursery.

Concurrent Garbage Collection

The concurrent garbage collection algorithm does its marking and sweeping “concurrently” with all other processing; that is, it does not stop Java threads to do the complete garbage collection.

Parallel Garbage Collection

The parallel garbage collection algorithm stops Java threads when the heap is full and uses every CPU to perform a complete mark and sweep of the entire heap. A parallel garbage collector can have longer pause times than concurrent garbage collectors, but it maximizes application throughput. Even on single CPU machines, this maximized performance makes parallel the recommended garbage collector, provided that your application can tolerate the longer pause times.

The Dynamic Garbage Collector

The dynamic garbage collector is the default garbage collector in JRockit and it combines the options and algorithms described above within the mark-and-sweep model to perform a garbage collection. Depending upon the heuristics used, the garbage collector will employ a two-generational or single-spaced collector with either a concurrent or parallel mark phase and a concurrent or parallel sweep phase.

The main benefit of a dynamic garbage collector is that the only determination you need to make for the best performance of your application is whether your application responds best to optimal throughput or minimized pause times during garbage collection. You do not need to understand the garbage collection algorithms themselves, or the various permutations thereof, just the behavior of your application.

To start the dynamic garbage collector, use the `-Xgcprio` command line option with either the `throughput` or `pausetime` parameter, depending upon which priority you want to use:

```
-Xgcprio:<throughput|pausetime>
```

If you set the pausetime option, you can also specify a target pause time for the garbage collection, for example:

```
-Xgcpausetarget=400ms
```

Table 3-1 describes the priorities under which you can start a dynamic garbage collector and the parameters used to select that priority.

Table 3-1 -Xgcprio Option Priorities

Priority	Description
Application Throughput (`-Xgcprio:throughput`)	The garbage collector is optimized for application throughput. This means that the garbage collector works as effectively as possible, giving as much CPU resources to the Java threads as possible. This may, however, cause non-deterministic pauses when the garbage collector stops all Java threads for garbage collection.The throughput priority should be used when non-deterministic pauses do not impact the application's behavior.
Pause Time (`-Xgcprio:pausetime`)	The garbage collector is optimized to limit the length of each garbage collection pause where all Java threads are stopped for garbage collection. This may result in lower application throughput, as the garbage collector uses more CPU resources in total than when running with the throughput priority. The pausetime priority should be used when the application depends on an even performance. Use `-Xpausetarget` to set a target length for the garbage collection pauses.

Upon selecting the priority and starting the JVM, the dynamic garbage collector will then try to choose the garbage collection state that optimizes performance based upon the priority. It will seek modes that optimize throughput when `-Xgcprio:throughput` is set or that minimize the pause times (as much as possible) when `-Xgcprio:pausetime` is set.

Using Backward-compatible Garbage Collectors

In some cases, you might not want to use a dynamic garbage collector. In those cases, you can specify one of the three static garbage collectors. The static garbage collectors will not attempt to optimize performance by changing algorithms. These garbage collectors are the original garbage collectors of earlier versions of JRockit. Depending on the circumstances, the performance of these collectors might meet your needs better than the dynamic garbage collector. Additionally, if you want to use scripts written for the earlier versions of JRockit that implement these collectors, those scripts will continue to work without requiring any modification—unless they use the copying garbage collectors, which are no longer available.

The available garbage collectors (and the command to start them) are:

- Single-spaced Concurrent (`-Xgc:singlecon`; this is the default garbage collector when BEA JRockit is run in the `-client` mode)
- Generational Concurrent (`-Xgc:gencon`)
- Parallel (`-Xgc:parallel`)

Overriding Garbage Collectors

Setting `-Xgc` will override `-Xgcprio` and any default settings.

Garbage Collector Selection Matrix

Table 3-2 is a matrix that you can use to determine which garbage collector is right for your application. Use the **If You...** column to locate a condition that matches what you want for your application and select the garbage collector indicated in the **Use this Garbage Collector...** column. The third column, **Or use...**, lists an alternate supported garbage collector.

Table 3-2 Garbage Collector Selection Matrix

If You...	Use this Garbage Collector...	Or use...
• Want to have as short pause times as possible. • Are willing to trade (some) application thoughput for shorter pauses. • Have a single CPU machine with a lot of memory. • Want better application throughput than possible with single-spaced concurrent.	`-Xgcprio:pausetime`	`-Xgc:singlecon` `-Xgc:gencon`
• Using a machine with four CPUs or better or a single CPU machine with a lot of memory. • Can tolerate the occasional long pause. • Need to maximize application throughput.	`-Xgcprio:throughput`	`-Xgc:parallel`
• Do not want a dynamic garbage collector.	`-Xgc:parallel`	`-Xgc:singlecon`

Viewing Garbage Collection Behavior

To observe garbage collection behavior, use one or all of the possibilities described here. Using this helps you evaluate the effectiveness of the selected garbage collector and makes it possible to make correct tuning decisions.

- If you want to view garbage collection during real-time, use the BEA JRockit Management Console tool. See the Using BEA JRockit Management Console document for information on how to use the tool.
- If you want information about the garbage collection during run-time, set the `-Xverbose:memory` option at startup. The information will appear in your console window.
- If you want to see a comprehensive report of garbage collection activity, enter the `-Xgcreport` option at startup. With this option JRockit prints a comprehensive garbage collection report when application run is completed.
- If you want to see the garbage collection activity when it occurs, enter the `-Xgcpause` option.This option causes the VM to print a line each time Java threads are stopped for garbage collection.

  You can combine the `-Xgcreport` and `-Xgcpause` at start up to examine the memory behavior of your application, for example, like this:

  ```
  java -Xgcreport -Xgcpause myClass
  ```

APPENDIX A

Using the Java Plug-in

Popular web browsers, such as Netscape Navigator and Microsoft Internet Explorer, can be connected to the Java platform by using the Java Plug-in, under which applets are run, already installed. If you want to run applets under BEA JRockit JRE, you can install the Java Plug-in that ships with this product.

Available only on ia32 platforms, the BEA JRockit Java Plug-in extends the functionality of your web browser, allowing applets and Java beans to run with JRockit. The Java Plug-in is part of the BEA JRockit JRE and is installed when the JRE is installed on a computer. It works with Netscape, Mozilla, and Internet Explorer.

This section includes information on the following subjects:

- Supported Operating Systems and Browsers
- Installing the Plug-in
- Plug-in Reference

Supported Operating Systems and Browsers

Table A-1 lists the operating systems and browsers supported by the BEA JRockit Java Plug-in.

Table A-1 Java Plug-in O/S and Browser

Operating System Support	For a list of supported operating systems, see the list of supported ia32 platforms at BEA JRockit JDK Platform Support.
Browser Support	Internet Explorer 5.5+; Netscape 4.5+ , 4.79*, Netscape 6.2+, and Mozilla 1.4+. **Note:** *Netscape 4.79 will only run applets that are specifically tagged to be run by 1.5.0.

Installing the Plug-in

The Java Plug-in is installed automatically for Windows machines when you install the BEA JRockit JRE, as described in "Installing the JRE." For Linux machines, you can install it as described in either of these documents from Sun Microsystems:

- Manual Installation/Registration of Java Plug-in—Linux (manual installation and registration)
- Control Panel Script Options for Plug-in Registration (automatic installation and registration)

Note on Installing the BEA JRockit Plug-in and Sun Plug-in

If you install the Sun JRE after installing the BEA JRockit JRE, the Sun JRE becomes the default Java Plug-in on the system. If this happens, you can uninstall and reinstall the BEA JRockit JRE.

Plug-in Reference

Generally, once the plug-in is installed, its behavior will be transparent and require little, if any, user intervention. However, there are many other related topics that you may want to understand. Sun Microsystems provides helpful information on the Java Plug-in that is fully compatible with the BEA JRockit Java plug-in. You can find this information at:

`http://java.sun.com/products/plugin/index.jsp`

Using Web Start with BEA JRockit

This version of BEA JRockit includes an implementation of Java Web Start, a tool that allows you to start Java applications with a single click in your browser. With Web Start, you can download and launch applications directly from the browser and avoid complex and time-consuming installation procedures. Any Java application can be started by using Web Start.

This section includes information on the following subjects:

Platform Support

Java Web Start is available only on Windows IA32 and Linux IA32 systems.

What You Can Do with Web Start

With Java Web Start, you launch applications simply by clicking on a Web page link. If the application is not present on your computer, Java Web Start automatically downloads all necessary files. It then caches the files on your computer so the application is always ready to be relaunched anytime you want—either from an icon on your desktop or from the browser link.

And no matter which method you use to launch the application, the most current version of the application is always presented to you.

Web Start Security

Java Web Start includes the security features of the Java platform to ensure the integrity of your data and files. It also enables you to use the latest Java 2 technology with any browser.

Installing and Launching Web Start

Java Web Start is installed as part of the public JRE installation (see Installing the BEA JRockit JRE).

Windows Platforms

Upon installation, a new icon will appear on your desktop (Figure B-1) and a new program group appears in your **Start** menu, under **Programs**.

Figure B-1 Java Web Start Icon

Use either of these to launch Java Web Start:

`<jre_home>/bin/javaws`

where `<jre_home>` is you JRE home directory, for example:

`C:\Program File\Java\jrockit-jdk1.5.0_02\jre`).

Linux Platforms

The Linux installation itself does not change with Web Start added; however, you can only launch Web Start from the command line. Do so by entering the command:

`<jre_home>/bin/javaws`

Note: JPackage RPMs will install Java Web Start and you can start by using the same command used for other Linux platforms.

Comprehensive Web Start Documentation

Java Web Start is a Sun Microsystems product and the BEA JRockit implementation is no different than Sun's. Please refer to the following documents for more complete information on using this feature:

- Java Web Start Developers Section:

http://java.sun.com/products/javawebstart/developers.html

- Java Web Start API Specification:

http://java.sun.com/products/javawebstart/reference/api/index.html

- Code Samples and Applications:

http://java.sun.com/products/javawebstart/reference/codesamples/index.html

- Technical Articles & Tips:

http://java.sun.com/products/javawebstart/reference/techart/index.html

- FAQs:

http://java.sun.com/products/javawebstart/faq.html

Using jstat with BEA JRockit

The JVM Statistics Monitoring Tool, **jstat**, attaches to a Java virtual machine and collects and logs performance statistics as specified by the command line options. This tool is developed by Sun Microsystems Inc. and it is included in the installation package of BEA JRockit. For a complete description on how jstat works, please refer to: http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/jstat.html. This appendix lists differences in the output when running the tool with BEA JRockit.

Statistics Options and Output

The following table lists all available statistical options, statOption, when using BEA JRockit. The tables that follow, summarize the columns that **jstat** outputs for each statOption.

statOption	Displays
`-class`	Statistics on the behavior of the class loader.
`-compiler`	Statistics of the behavior of the JRockit compiler.
`-gc`	Statistics of the behavior of the garbage collected heap.
`-gcpause`	Statistics of garbage collection pauses.
`-printjit`	Statistic on the latest JIT compiler operations.
`-printopt`	Statistic on the latest optimizing compiler operations.

-class statOption

Column	Description
Loaded	Number of classes loaded.
Bytes	Number of Kbytes loaded.
Unloaded	Number of classes unloaded.
Bytes	Number of Kbytes unloaded.

-compiler statOption

Column	Description
Compiled	Number of compilation tasks performed.
Jitted	Number of JIT compilation tasks performed.
Optimized	Number of optimization compilation tasks performed.
Time	Time spent performing compilation tasks.
TotalTimeJit	Time spent performing JIT compilation tasks.
TotalTimeOpt	Time spent performing optimizing compilation tasks.

-printjit statOption

Column	Description
Jitted	Number of JIT compilation tasks performed.
TotalTimeJit	Time spent performing JIT compilation tasks.
LastJitMethod	The last method compiled by the JIT.

-printopt statOption

Column	Description
`Optimized`	Number of optimization compilation tasks performed.
`TotalTimeOpt`	Time spent performing optimizing compilation tasks.
`LastOptMethod`	The last method compiled by the optimizing compiler.

-gc statOption

Column	Description
`HeapSize`	Heap size (KB).
`NurserySize`	Nursery size (KB).
`UsedHeapSize`	Used heap size (KB).
`YC`	Number of young generation collections.
`OC`	Number of old generation collections.
`YCTime`	Young generation garbage collection time (s).
`OCTime`	Old generation garbage collection time (s).
`GCTime`	Total garbage collection time (s).
`YCPauseTime`	Young generation pause time (s).
`OCPauseTime`	Old generation pause time (s).
`PauseTime`	Total pause time (s).
`Finalizers`	Number of pending finalizers.

-gcpause statOption

Column	Description
YC	Number of young generation collections.
OC	Number of old generation collections.
YCPauseTime	Young generation pause time (s).
OCPauseTime	Old generation pause time (s).
PauseTime	Total pause time (s).

Monitoring Thread Activity With Thread Dumps

Thread dumps, or "thread stack traces," reveal information about an application's activity that can help you diagnose problems and better optimize application and JVM performance; for example, thread dumps can show the occurrence of "deadlock" conditions, which can seriously impact application performance.

You can create a thread dump by invoking a control break (usually by pressing `Ctrl-Break` or `Ctrl-\` or `SIGQUIT` on linux). This section provides information on working with thread dumps. It includes information on these subjects:

Lock Information in Thread Dumps

When printing stack traces with `Control-Break` or `SIGQUIT` on linux, BEA JRockit also shows the status of active locks (monitors). For each thread, BEA JRockit prints the following information if the thread is in a waiting state:

- If the thread is trying to take a lock (to enter a synchronized block), but the lock is already held by another thread, this is indicated at the top of the stack trace, as "Blocked trying to get lock".
- If the thread is waiting for a notification on a lock (by calling `Object.wait()`), this is indicated at the top of the stack trace as "Waiting for notification".

- If the thread has taken any locks, this is shown in the stack trace. After a line in the stack trace describing a function call is a list of the locks taken by the thread in that function. This is described as `^-- Holding lock` (where the `^--` serves as a reminder that the lock is taken in the function written above the line with the lock).

Caution: The lines with the lock information might not always be correct, due to compiler optimizations. This means two things:

- If a thread, in the same function, takes first lock A and then lock B, the order in which they are printed is unspecified.
- Sometimes, if a thread, in method `foo()` calls method `bar()`, and takes a lock A in `bar()`, the lock might be printed as being taken in `foo()`.

Normally, this shouldn't be a problem. The order of the lock lines should never move much from their correct position. Also, lock lines will never be missing—you can be assured that all locks taken by a thread are shown in the stack dump.

The semantics for waiting (for notification) on an object in Java is somewhat complex. First you must take the lock for the object, and then you call `wait()` on that object. In the wait method, the lock is released before the thread actually goes to sleep waiting for a notification. When it receives a notification, wait re-takes the lock before returning. So, if a thread has taken a lock, and is waiting (for notification) on that lock, the line in the stack trace that describes when the lock was taken is not shown as “Holding lock,” but as “Lock released while waiting.”

All locks are described as `Classname@0xLockID[LockType]`; for example:

```
java/lang/Object@0x105BDCC0[thin lock]
```

Where:

- `Classname@0xLockID` describe the object the to which the lock belongs. The classname is an exact description, the fully qualified class name of the object. `LockID`, on the other hand, is a temporary ID which is only valid for a single thread stack dump. That is, you can trust that if a thread A holds a lock `java/lang/Object@0x105BDCC0`, and a thread B is waiting for a lock `java/lang/Object@0x105BDCC0`, in a single thread stack dump, then it is the same lock. If you do any subsequent stack dumps however, `LockID` is not comparable and, even if a thread holds the same lock, it might have a different `LockID` and, conversely, the same `LockID` does not guarantee that it holds the same lock.
- `LockType` describes the kind of BEA JRockit internal lock type the lock is. Currently, three kinds of locks exist:
  - **fat locks**: locks with a history of contention (several threads trying to take the lock simultaneously), or that have been waited on (for notification).

- **thin locks**: locks that have had no contention.
- **recursive locks**: locks occur when a thread takes a lock it already holds.

Listing 3-1 shows an example of what a stack trace for a single thread can look like.

Listing 3-1 Example: Stack Trace for a Single Thread

```
"Open T1" prio=5 id=0x680 tid=0x128 waiting
   -- Waiting for notification on: java/lang/Object@0x1060FFC8[fat lock]
  at jrockit/vm/Threads.waitForSignalWithTimeout(Native Method)@0x411E39C0
   at jrockit/vm/Locks.wait(Locks.java:1563)@0x411E3BE5
   at java/lang/Thread.sleep(Thread.java:244)@0x41211045
   ^-- Lock released while waiting: java/lang/Object@0x1060FFC8[fat lock]
   at test/Deadlock.loopForever(Deadlock.java:67)@0x412304FC
   at test/Deadlock$LockerThread.run(Deadlock.java:57)@0x4123042E
   ^-- Holding lock: java/lang/Object@0x105BDCC0[recursive]
   ^-- Holding lock: java/lang/Object@0x105BDCC0[thin lock]
   at java/lang/Thread.startThreadFromVM(Thread.java:1690)@0x411E5F73
   --- End of stack trace
```

Detecting Deadlocks

After the normal stack dumps, BEA JRockit performs a deadlock detection. This is done by finding “lock chains” in the Java application. If a lock chain is found to be circular, the application is considered caught in a deadlock.

What is a “Lock Chain”?

Although they appear somewhat complex, lock chains are fairly straightforward. Informally, lock chains can be described as a sequence of threads, each waiting for a lock held by the next thread in the chain. An open lock chain ends with a thread that is not trying to take a lock, but is instead doing actual work or possibly waiting for some external event. A circular chain is a deadlock—it will never be resolved. A closed chain depends on another lock chain, and is in effect deadlocked if the other chain is deadlocked, and open if the other chain is open. Closed chains mean that several threads are trying to take the same lock.

Formal Definition of a Lock Chain

1. If thread `Tb` holds lock `Lb`, and thread `Ta` is trying to take lock `Lb`, then they form the **lock chain** `Ta->Tb`.
2. If `Ta->Tb` is a lock chain, and thread `Tc` is holding the lock `Lc`, which thread `Tb` is trying to take, then `Ta->Tb->Tc` is also a lock chain.
3. If `Ta->...->Tn` is a lock chain, and there exist no lock `La` held by `Ta` and a thread `Tx` such that `Tx` is trying to take `La`, then the lock chain is **starting at** `Ta`.
4. If `Ta->...->Tn` is a lock chain starting at `Ta`, and there exist no lock `Lx` such that thread `Tn` is trying to take `Lx`, then `Ta->...->Tn` is an **open** lock chain, **ending on** `Tn`.
5. If `Ta->...->Tn` is a lock chain starting at `Ta`, and thread `Tn` is trying to take lock `Lo`, which is held by a thread `To`, and thread `To` is involved in a separate, complete lock chain, then `Ta->...->Tn` is a **closed** lock chain, **ending on** `Tn`.
6. If `Ta->...->Tn` is a lock chain, and thread `Tn` is trying to take lock `La` held by thread `Ta`, then `Ta->...->Tn` is a **circular** (deadlocked) lock chain.
7. A lock chain is **complete** if it is either an open, closed or circular lock chain.

From the definitions follows that all threads that are trying to take a lock belong to exactly one complete lock chain

Lock Chain Dump

BEA JRockit will find all complete lock chains, and will group them into open, closed and circular lock chains. All open and closed lock chains will be printed from their starting elements to their end. Circular lock chains has neither a start nor an end—BEA JRockit will chose an element arbitrarily and treat it like the start.

The division between a closed lock chain and the other lock chain is arbitrary. Closed chains arise whenever two different threads are blocked trying to take the same lock; for example: `Thread A` holds lock `Lock A` while `Thread B` is waiting for `Lock A`; `Thread C` is also waiting for `Lock A`. BEA JRockit will interpret this in one of the following ways:

- `B > A` as an open lock chain and `C > A` as a closed lock chain.
- `C > A` as an open lock chain and `B > A` as a closed lock chain.

A deadlocked lock chain can never be resolved, and the application will be stuck waiting indefinitely. If you have long (but open) lock chains, your application might be spending unnecessary time waiting for locks.

Index

X