Oracle® JRockit

Introduction

Release R28

E15058-04

April 2011

This document contains information about the Oracle JRockit JDK

ORACLE®

Oracle JRockit Introduction, Release R28

E15058-04

Primary Author: Savija T.V.

Contents

6 Setting Up and Running the Oracle JRockit JDK

Preface

This preface describes the document accessibility features and conventions used in this guide—*Oracle JRockit Introduction*.

Guide to this Document

This document contains the following chapters:

- Chapter 1, "About the Oracle JRockit JDK" provides an overview of the Oracle JRockit JDK.
- Chapter 2, "Understanding Just-In-Time Compilation and Optimization" describes how Oracle JRockit compiles and optimizes the Java code.
- Chapter 3, "Understanding Threads and Locks" provides information about threads and locks in the Oracle JRockit JVM.
- Chapter 4, "Understanding Memory Management" describes how object allocation and garbage collection work in the Oracle JRockit JVM.
- Chapter 5, "Migrating Applications to the JRockit JDK" describes how to migrate Java applications developed on other JDKs to the JRockit JDK.
- Chapter 6, "Setting Up and Running the Oracle JRockit JDK" describes the process of setting up the Oracle JRockit JDK environment.

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.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

About the Oracle JRockit JDK

The Oracle JRockit provides tools, utilities, and a complete run-time environment for developing and running Java applications. The JRockit includes the Oracle JRockit JVM, which is developed and optimized for Intel architectures to ensure reliability, scalability, and manageability for Java applications.

This chapter contains information about the following topics:

1.1 What Is the JRockit JVM?

The JRockit JVM is a high-performance JVM developed to ensure reliability, scalability, manageability, and flexibility for Java applications. The JRockit JVM provides improved performance for Java applications deployed on Intel 32-bit (Xeon) and 64-bit (Xeon and SPARC) architectures at significantly lower costs to the enterprise. Further, it is the only enterprise-class JVM optimized for Intel architectures, providing seamless interoperability across multiple hardware and operating system configurations. The JRockit JVM enables your Java applications to run optimally on Windows and Linux operating systems (both 32-bit and 64-bit architectures). The JRockit JVM is especially well-suited for running Oracle WebLogic Server.

For more information about JVMs, see *The Java Virtual Machine Specification* at: `http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html`

About the Oracle JRockit JDK

The JRockit JVM is one component of the JRockit JDK. In addition to the JRockit JVM, the JDK contains the Java Run-time Environment (JRE). The JRE contains the Java class libraries (as specified by the Java Platform, Standard Edition 6 API Specification) and a set of development tools, such as a compiler.

For more information about the contents of the JRockit, see Section 1.6, "Contents of a JRockit Installation".

1.2 JRockit Release Numbering Scheme

Every JRockit JVM release comes with several Java versions. For example, JRockit JVM R28.0 comes with Java SE versions 5.0 and 6. A Java version can be compatible with multiple JRockit JVM releases.

The JRockit release number consists of the following elements:

- The JRockit JVM release number (R*nn.nn.nn*)
- The Java version (J2SE 5.0 or Java SE 6)

For example, **Oracle JRockit 6 R28.0.0** indicates the 28.0.0 release of JRockit JVM used with Java SE 6; similarly, **Oracle JRockit 5.0 R28.0.0** indicates the 28.0.0 release of the JRockit JVM used with J2SE 5.0.

The following is an example of a complete release number:

```
R28.0.0-637-126675-1.6.0_17-20100111-2121-windows-ia32
```

In this example, `R28.0.0` is the JRockit JVM release, `1.6.0_01` is the Java version, and `windows-ia32` is the platform on which the release runs.

1.3 Supported Configurations for the Oracle JRockit

The JRockit is compatible with J2SE 5.0 and Java SE 6.

For a full list of processor types and operating systems that the JRockit supports, see *Oracle JRockit Supported Configurations* at http://www.oracle.com/technology/software/products/ias/files/fusion_certification.html.

1.4 Support and Error Correction Policies

The *Oracle Lifetime Support Policy* for Oracle Fusion Middleware is available at http://www.oracle.com/support/library/brochure/lifetime-support-middleware.pdf.

The *Software Error Correction Policy* is available in Support Note ID 950131.1.

1.5 Compatibility Between JRockit and HotSpot

Java APIs

All Java APIs follow the Java compatibility statement (http://java.sun.com/javase/6/webnotes/compatibility.html).

Proprietary JRockit JVM APIs

- Official APIs (JMAPI, JRockit JMX beans) cannot be removed or modified except between major JDK versions. New APIs can be added at any time.
- Unofficial but supported APIs (`jrockit.ext.*`) cannot be removed in service packs.
- Internal APIs (all `jrockit.*` except for `jrockit.ext.*`) can change at any time.

Command Line Options

- Standard options (for example, `-server`) adhere to the Java compatibility statement (http://java.sun.com/javase/6/webnotes/compatibility.html).
- Nonstandard `-X` options adhere to the Java compatibility statement (http://java.sun.com/javase/6/webnotes/compatibility.html).
- Nonstandard `-XX` options are handled as follows:
  - They cannot be removed unless in association with major JDK updates.
  - They can be disabled or their implementation can be changed at any time.
  - New `-XX` options can be added at any time.

1.6 Contents of a JRockit Installation

The JRockit is similar, in the file layout, to the HotSpot JDK, except that the JRockit includes a new JRE with the JRockit JVM. All of the class libraries have the same behavior in the JRockit as in the HotSpot JDK.

Table 1–1 describes the contents of the directories in a JRockit installation.

Table 1–1 Contents of a JRockit JDK Installation

Directory	Contents
`bin`	The `bin` directory contains development tools and utilities to help you develop, execute, debug, and document programs written in the Java programming language. The JRockit includes the standard tools distributed with the typical Java JDKs. Most of these tools work well with Java development projects, but you can use any other third-party tools, compilers, debuggers, IDEs, and so on that work best in your situation. The tools included with the JRockit are: ■ the **`javac`** compiler ■ the **`jdb`** debugger ■ **`javadoc`**, which is used to create documentation in HTML format for the JVM API For more information about these tools, see the documentation at http://java.sun.com/.
`demo`	The `demo` directory contains demos on using the libraries included in the JRockit JDK installation.
`include`	The `include` directory contains header files that support native-code programming using the Java Native Interface (JNI) and the Java Virtual Machine Tools Interface (JVMTI), and other functionality of the Java SE Platform.

Table 1–1 (Cont.) Contents of a JRockit JDK Installation

Directory	Contents

`jre`	The `jre` directory contains the JRockit JVM implementation of the Java run-time environment. The run-time environment includes the JRockit JVM, class libraries, and other files that support running Java programs.
	
	Java Virtual Machine: By definition, the JVM is the JRockit JVM, as described in this documentation.
	
	Standard Java SE JRE Features: In addition to JRE components specific to the JRockit, the JRE also contains components found in the HotSpot implementation of the JRE. For a complete list of the standard Java SE JRE features, see the following documentation:
	
	■ JRockit 6 R28.x:
	`http://java.sun.com/javase/6/docs/index.html`
	
	■ JRockit 5.0 R28.x:
	`http://java.sun.com/j2se/1.5.0/docs/index.html`
	
	Note about JRE class files: The JRE class files distributed with the JRockit are the same as those in the HotSpot JDK, except for a small number of files that are tightly coupled to the JVM and are, therefore, overridden in the JRockit. No classes are omitted.

`lib`	The `lib` directory contains additional class libraries and support files required by the development tools.
`sample`	The `sample` directory contains the source files for a simple NIO-based HTTP/HTTPS Server written in Java. The server was written to demonstrate some features of the Java 2 platform. The demo is not meant to be a full tutorial, and it is assumed that you have some familiarity with the subject.

1.7 JRockit JVM Command-Line Options

The JRockit JVM configuration and tuning parameters are set by using specific command-line options, which you can enter either along with the start-up command or include in a start-up script.

For more information, see the *Oracle JRockit Command-Line Reference*.

2

Understanding Just-In-Time Compilation and Optimization

This chapter offers a high-level look at how the Oracle JRockit JVM generates code. It provides information about how the JRockit JVM compiles code just-in-time (JIT) and how it optimizes code to ensure high performance.

This chapter contains information about the following topics:

- Section 2.1, "What Happens Inside the JRockit JVM"
- Section 2.2, "How the JRockit JVM Generates Machine Code for Java Applications"
- Section 2.3, "Examples of Code Optimization"

2.1 What Happens Inside the JRockit JVM

From the user's point of view, the JRockit JVM is a black box that converts Java code to highly optimized machine code for a specific platform (see Figure 2–1).

Figure 2–1 The JRockit JVM as a Black Box

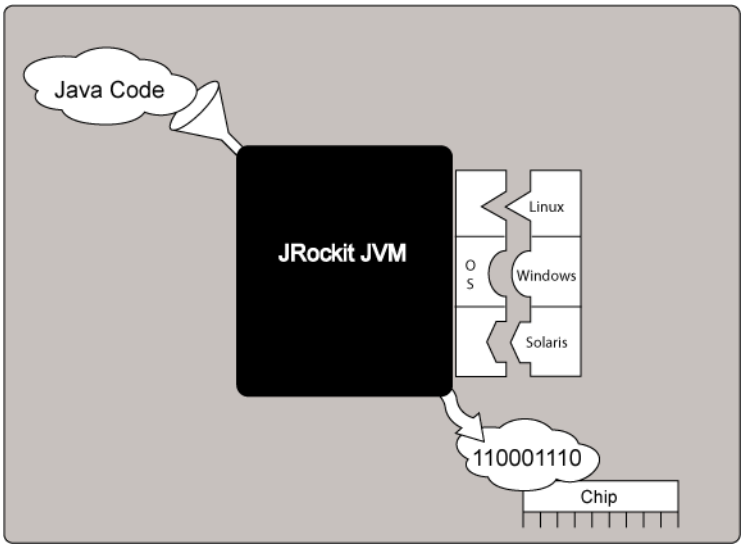
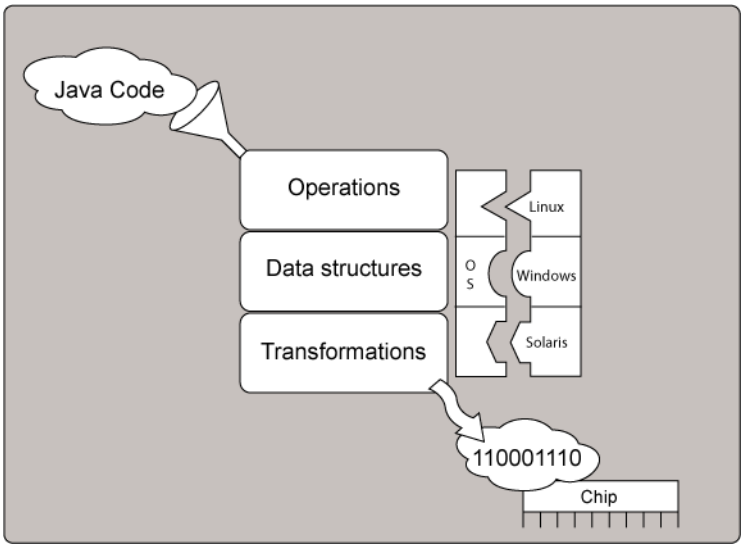


Figure 1-1 is a representation of the JRockit JVM as a black box. Java code enters the black box and emerges from it as binary code on a microchip. Along the right edge of the black box are the three operating systems (Linux, Windows, and Solaris) that the JRockit JVM supports.

****

Figure 2–2 shows what happens to Java code as it passes through the JRockit JVM. After Java code enters the JRockit JVM, it passes through the operations stage, the data structure stage, and the transformation stage, before emerging as binary code on a microchip.

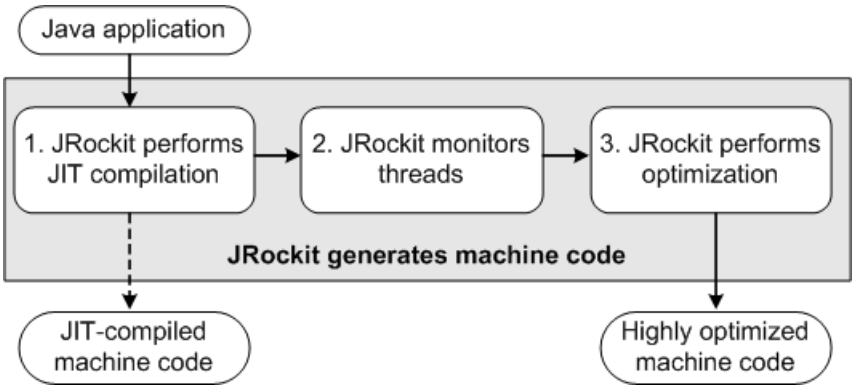
Figure 2–2 Inside the JRockit JVM Black Box



2.2 How the JRockit JVM Generates Machine Code for Java Applications

The code generator in the JRockit JVM runs in the background during the entire run time of your Java application, automatically adapting the code to run optimally. The code generator works in three steps, as depicted in Figure 2–3.

Figure 2–3 How the JRockit JVM Generates Machine Code for Your Java Application



Step 1: JIT Compilation

The first step in the machine-code generation process is just-in-time (JIT) compilation. JRockit does not include an interpreter; so the JIT compilation of the byte code into native machine code has to occur before a method executes. The JIT compilation is performed the first time a Java method is called.

The JIT compiler is fast and generates moderately efficient code. This is necessary to enable the Java application to start and run quickly. Subsequently, profiling reveals frequently called methods (hot spots) that require further optimization. The JRockit approach—JIT compilation and no interpreter—results in relatively longer startup times, but even if the JIT compilation results in only moderately efficient code, the generated code is still significantly faster than interpreted code.

Step 2: Thread Monitoring

During the second step of the machine-code generation process, the JRockit JVM uses a sophisticated, low-cost, sampling based technique to identify which functions merit optimization: a "sampler thread" wakes up periodically and checks the status of several application threads. It identifies what each thread is executing and logs some execution history. This information is tracked for all the methods; when the information indicates that a method is heavily used (hot), that method is earmarked for optimization. Usually, a flurry of such optimization opportunities occur in the application's early run stages, with the rate slowing down as execution continues.

Step 3: Code Optimization

Code optimization is a process by which commonly-executed code is recompiled to make it run more efficiently.

The first time that the JRockit JVM runs a method, the method is compiled into machine code. This compilation is quick, but the resulting code is not as efficient as it could be. This code is acceptable for methods that are run once and discarded; however, if a method is used repeatedly, the system can get a performance boost if the code for that particular method is regenerated in a more efficient way.

The JRockit JVM optimizes these commonly-executed (hot) methods to make the code as efficient as possible. The optimization runs in the background and does not interfere with the running application.

2.3 Examples of Code Optimization

The following code examples show how the JRockit JVM optimizes Java code.

Example 2–1 shows the code before optimization.

Example 2–1 Code Before Optimization

```
class A {
  B b;
  public void newMethod() {
    y = b.get();
    ...do stuff...
    z = b.get();
    sum = y + z;
  }
}
class B {
   int value;
   final int get() {
      return value;
   }
}
```

Example 2–2 shows the optimized code.

Example 2–2 Code After Optimization

```
class A {
B b;
public void newMethod() {
   y = b.value;
   ...do stuff...
   sum = y + y;
}
}
class B {
   int value;
   final int get() {
      return value;
   }
}
```

Originally, the code contained two calls to the `b.get()` method. After optimization, the two method calls are optimized into a single variable-copy operation; that is, the optimized code does not need to perform a method call to acquire the field value of class B.

Optimization Process: Step-by-Step

Table 2–1 shows the tasks that the JRockit JVM performs to optimize the code in Example 2–1.

Note that the optimization process usually does not occur at the Java source code level. Java source code is used here to provide an easily understandable view of the optimization process.

Table 2–1 Optimization Process: Step-by-Step

Optimization Step	Code Transformation	Comment

Starting point	`public void newMethod() {`
	`  y = b.get();`
	`  ...do stuff...`
	`  z = b.get();`
	`  sum = y + z;`
	`}`	

Inline final method	`public void newMethod() {`
	`  y = b.value;`
	`  ...do stuff...`
	`  z = b.value;`
	`  sum = y + z;`
	`}`
		`// b.get() is replaced by b.value`
		`// latencies are reduced by accessing`
		`// b.value directly instead of using`
		`// a function call.`

Remove redundant loads	`public  void newMethod()`
	`{`
	`  y = b.value;`
	`  ...do stuff...`
	`  z = y;`
	`  sum = y + z;`
	`}`
		`// z = b.value is replaced with`
		`// z = y so that latencies are`
		`// reduced by accessing local value`
		`// instead of b.value.`

Table 2–1 (Cont.) Optimization Process: Step-by-Step

Optimization Step	Code Transformation	Comment

Copy propagation	`public void newMethod()`
	`{`
	`  y = b.value;`
	`  ...do stuff...`
	`  y = y;`
	`  sum = y + y;`
	`}`
		`// z = y is replaced by y = y because`
		`// there is no use for extra variable`
		`// z; the values of z and y are equal.`

Eliminate dead code	`public void newMethod()`
	`{`
	`  y = b.value;`
	`  ...do stuff...`
	`  sum = y + y;`
	`}`
		`// y = y is unnecessary and can be`
		`// eliminated.`

3 Understanding Threads and Locks

A running application usually consists of one process with its own memory space. A computer generally runs several processes at the same time. For example, a word processor application process might run at the same time as a media player application process. A process consists of many threads that run concurrently. When you run a Java application, a new JVM process starts.

Each Java process has at least one application thread. In addition to the application threads, the Oracle JRockit JVM uses internal threads for garbage collection, code generation, and other internal purposes.

You can use thread dumps (printouts of the stacks of all the threads in an application) to diagnose problems and optimize application and JVM performance. For information about generating thread dumps, see "Using Thread Dumps" in *Oracle JRockit JDK Tools*.

The following sections contain basic information about threads and locks in the JRockit JVM.

3.1 Understanding Threads

A Java application consists of one or more threads that run Java code. The JVM process consists of the Java threads and some JVM-internal threads, for example one or more garbage collection threads, a code optimizer thread, and one or more finalizer threads.

The operating system handles Java threads as it does any application thread. Thread scheduling and priorities are handled by the operating system.

Within Java, the Java threads are represented by thread objects. Each thread also has a stack, used for storing run-time data. The thread stack has a specific size. If a thread tries to store more items on the stack than the stack size allows, the thread throws a stack overflow error.

Default Stack Sizes for Java Application Threads

The default stack sizes vary depending on the operating system you use. Table 3–1 lists the default stack sizes for different operating systems.

Table 3–1 Default Stack Size

Operating System	Default Stack Size
Windows IA32	64 KB
Windows x64	128 KB
Linux IA32	128 KB
Linux x64	256 KB
Solaris/SPARC	512 KB

You can change the thread stack size with the `-Xss` command-line option (for example, **java -Xss:512k MyApplication**

Default Stack Size for JVM Internal Threads

A special "system" stack size (256 KB on all platforms) is used for JVM internal threads such as the garbage collection and code generation threads.

> **Note:** The `-Xss` command-line option sets the stack size of both application threads and JVM internal threads.

3.2 Understanding Locks

When threads in a process share and update the same data, their activities must be synchronized to avoid errors. In Java, this is done with the `synchronized` keyword, or with the `wait` and `notify` keywords. Synchronization is achieved by the use of locks, each of which is associated with an object. For a thread to work on an object, it must control (hold) the associated lock. Only one thread can hold a lock at a time. If a thread tries to take control of a lock that is already held by another thread, then it must wait until the lock is released. When this happens, there is contention for the lock.

Locks can be of four types:

- Thin locks

  A thread that tries to acquire a lock that is held by another thread **spins** for a short while. While it spins, the thread continuously checks whether the lock it needs is still held by the other thread. This is the default behavior on multiple-CPU systems. Such a lock is called a **thin** lock.

- Fat locks

  If a thread trying to acquire a lock spins for too long, the CPU resource is needlessly tied down and not available to execute other code. The CPU resource can be released, by inflating the thin lock into a **fat** lock. The operating system then manages the thread till the required lock becomes available, freeing the CPU for other tasks.

- Recursive locks

  A lock is recursive if the synchronized code calls itself, either directly or indirectly.

- Lazy locks

  A lazy lock is not released when a critical section is exited. After a lazy lock is acquired by a thread, other threads that try to acquire that lock must ensure that the lock is, or can be, released.

The JRockit JVM uses a complex set of heuristics to determine when to change from one type of lock to another.

Lock Chains

Several threads can be tied up in what is called a lock chain. A lock chain can be defined as follows:

- Threads A and B form a lock chain if thread A holds a lock that thread B is needs. If A is not trying to take a lock, then the lock chain is considered to be open.
- If A–B is a lock chain, and B–C is a lock chain, then A–B–C is a more complete lock chain.
- If there is no additional thread waiting for a lock held by C, then A–B–C is a complete and open lock chain.

Lock Chain Types

The JRockit JVM analyzes the threads and forms complete lock chains. There are three possible types of lock chains: open, deadlocked and blocked.

- Open chains

  Open chains represent a straight dependency: thread A is waiting for B which is waiting for C, and so on. If you have long open chains, your application might be wasting time waiting for locks. If this is the case, then reconsider how locks are used for synchronization in your application.

- Deadlocked chains

  A deadlocked, or circular, chain consists of a chain of threads, in which the first thread in the chain is waiting for the last thread in the chain. In the simplest case, thread A is waiting for thread B, while thread B is waiting for thread A. In thread dumps, the JRockit JVM selects an arbitrary thread to display as the first thread in the chain.

  Deadlocks cannot be resolved, and the application waits indefinitely.

- Blocked chains

  A blocked chain is a lock chain whose head thread is also part of another lock chain, which can be either open or deadlocked. For example, if thread A is waiting for thread B, thread B is waiting for thread A, and thread C is waiting for thread A, then thread A and B form a deadlocked lock chain, while thread C and thread A form a blocked lock chain.

4

Understanding Memory Management

Memory management is the process of removing unused objects from the **heap**, to make memory available for new objects. This chapter presents basic memory management concepts and explains how object allocation and garbage collection work in the Oracle JRockit JVM. It contains information about the following topics:

> **Note:** This chapter is primarily about memory management theory. For information about how to tune the memory management system, see "Tuning the Memory Management System" in the *Oracle JRockit Performance Tuning Guide*.

4.1 Heap and Nursery

Java objects reside in an area called the heap, a section of the memory stack that is created when the JVM starts. The heap can grow or shrink while the application runs.

When the heap becomes full, **garbage** is **collected**: The JRockit JVM identifies memory spaces that contain objects that are being used (live objects). It then reclaims the memory spaces that do not contain live objects, and makes those spaces available for allocation to new objects.

> **Note:** The JRockit JVM uses more memory than the heap; Java methods, thread stacks, native handles, and JVM internal data structures are allocated in a memory space that is separate from the heap.

The heap is sometimes divided into two **generations**: a **nursery** (or **young space**) and an **old space**. The nursery is a part of the heap reserved for allocating new objects. When the nursery becomes full, garbage is collected by running a special **young collection**, through which all objects that have lived long enough in the nursery are **promoted** (moved) to the old space, freeing up the nursery for more object allocation. When the old space is full, garbage is collected there—a process called **old collection**.

Note: At times, during a young collection, there might not be enough memory left in the old space to promote objects from the nursery. Until memory becomes available in the old space, objects that are identified for promotion to the old space are instead promoted within the nursery, resulting in a fragmented nursery. This situation is known as promotion failure.

In R28.1, the JRockit JVM assesses the likelihood of an upcoming young collection resulting in a promotion failure, and prevents it by triggering an old collection.

A multigenerational heap helps reduce the overall garbage collection time. Most objects allocated in the heap are short-lived. The young collection quickly finds newly allocated objects in the nursery that are still being used and moves them out of the nursery to the old space. Typically, a young collection frees a given amount of memory much faster than an old collection or a garbage collection of a single-generational heap (a heap without a nursery).

A part of the nursery is reserved as a **keep area**, for holding objects that were allocated just before a young collection started. These objects are not garbage collected until the next young collection.

4.2 Object Allocation

During object allocation, the JRockit JVM distinguishes between small and large objects. The limit for when an object is considered large depends on the heap size, the garbage collection strategy, and the platform being used, but is usually between 2 and 128 KB. For more information, see the documentation for the `-XXtlaSize` command-line option in the *Oracle JRockit Command-Line Reference*.

Small objects are allocated in **thread local areas** (TLAs), which are free chunks of memory reserved from the heap and given to a Java thread for exclusive use. The thread can then allocate objects in its TLA without synchronizing with other threads. When the TLA becomes full, the thread requests for a new TLA. The TLAs are reserved in the nursery, if one exists; otherwise, they are reserved anywhere on the heap.

Objects that are larger than the TLA size are allocated directly on the heap. Allocation of large objects requires more synchronization among the Java threads; the JRockit JVM uses a system of caches of free chunks of different sizes to reduce the need for synchronization and to improve the allocation speed.

4.3 Garbage Collection

Garbage collection frees space in the heap for allocating new objects.

The following topics describe how garbage collection works in the JRockit JVM.

- Section 4.3.1, "Mark-and-Sweep Model"
- Section 4.3.2, "Generational Garbage Collection"
- Section 4.3.3, "Garbage Collection Modes"

4.3.1 Mark-and-Sweep Model

The JRockit JVM uses the mark-and-sweep garbage collection model to perform garbage collections of the whole heap.

- During the **mark phase**, all objects that can be reached directly from Java threads, native handles, and other root sources; objects that can be reached from the first set of objects; and so on are marked as being used (live objects). The remaining objects are considered garbage.
- During the **sweep phase**, the heap is traversed to identify the gaps between the live objects. The gaps are recorded in a **free list** and are made available for new object allocation.

The JRockit JVM provides two improved strategies of the mark-and-sweep model: **mostly concurrent** and **parallel**. You can mix the two strategies; for example, you can run the mostly concurrent mark with a parallel sweep.

Mostly Concurrent Mark-and-Sweep Strategy

The mostly concurrent mark-and-sweep strategy (often called concurrent garbage collection) allows the Java threads to continue running during large portions of the garbage collection process. The threads must, however, be stopped a few times for synchronization.

The mostly concurrent mark phase is divided into four stages:

- Initial marking

  The root set of live objects is identified. This is done while the Java threads are paused.

- Concurrent marking

  The references from the root-set are followed to find and mark the remaining live objects in the heap. This is done while the Java threads run.

- Precleaning

  Changes in the heap during the concurrent mark phase are identified and any additional live objects are found and marked. This is done while the Java threads run.

- Final marking

  Changes during the precleaning phase are identified and any additional live objects are found and marked. This is done while the Java threads are paused.

The mostly concurrent sweep phase consists of four stages:

- Sweeping one half of the heap

  This occurs while the Java threads run; object allocation can continue in the part of the heap that is currently not being swept.

- A short pause to switch halves
- Sweeping the other half of the heap

  This is done while the Java threads run; object allocation can continue in the part of the heap that was swept first.

- A short pause for synchronization and recording statistics

Parallel Mark-and-Sweep Strategy

The parallel mark and sweep strategy (also called the parallel garbage collector) uses all available CPUs in the system for performing the garbage collection as fast as possible. All Java threads are paused during the parallel garbage collection.

4.3.2 Generational Garbage Collection

When a nursery exists (as described in Section 4.1, "Heap and Nursery"), its garbage is collected with a special garbage collection called a young collection. A garbage collection strategy that uses a nursery is called generational garbage collection.

The young collector used in the JRockit JVM identifies and promotes all live objects in the nursery that are outside the keep area to the old space. This work is done in parallel by using all available CPUs. The Java threads are paused during the young collection.

4.3.3 Garbage Collection Modes

The JRockit JVM automatically selects a garbage collection strategy that optimizes the application throughput.

The following garbage collection modes are available:

- **throughput** (default mode): Optimizes the garbage collector for maximum application throughput
- **pausetime**: Optimizes the garbage collector for short and even pause times
- **deterministic** (only available as a part of Oracle JRockit Real Time): Optimizes the garbage collector for very short and deterministic pause times

For more information, see "Selecting and Tuning a Garbage Collector" in the *Oracle JRockit Performance Tuning Guide*.

4.4 Compaction

After a garbage collection, the heap may become fragmented. Numerous free spaces exist, but each free space is small, making allocation of large objects difficult or impossible. Free spaces that are smaller than the minimum TLA size cannot be used and the garbage collector discards them as dark matter until a future garbage collection frees space next to them, creating a space large enough for a TLA.

To reduce fragmentation, the JRockit JVM compacts a part of the heap at every garbage collection (old collection). Compaction moves objects closer and further down in the heap, creating larger free areas near the top of the heap. The size (and position) of the compaction area and the compaction method are selected by advanced heuristics, depending on the garbage collection mode used.

Compaction is performed at the beginning of or during the sweep phase and while all Java threads are paused.

For information about how to tune compaction, see "Tuning the Compaction of Memory" in the *Oracle JRockit Performance Tuning Guide*.

External and Internal Compaction

The JRockit JVM uses two compaction methods: external compaction and internal compaction.

External compaction moves (evacuates) the objects within the compaction area to free positions outside the compaction area and as far down in the heap as possible. Internal compaction moves the objects within the compaction area as far down in the compaction area as possible, moving them closer.

The JVM selects a compaction method depending on the current garbage collection mode and the position of the compaction area. External compaction is typically used near the top of the heap, while internal compaction is used near the bottom, where the density of objects is higher.

Compaction Area Size and Location

The size and location of the compaction area depends on the garbage collection mode and the current state of the Java heap.

If the number of references to objects within an area (object density) is high, the compaction area size is small. Typically the object density is higher toward the bottom of the heap than toward the top, except at the very top of the heap, which contains recently allocated objects. So the compaction areas are usually smaller near the bottom of the heap than at the top of the heap.

5

Migrating Applications to the JRockit JDK

This chapter describes how to migrate Java applications developed on other JDKs to the JRockit JDK to ensure optimal performance during run time. It contains the following topics:

> **Note:** Successful migration to the Oracle JRockit JDK often depends as much on the application being migrated as on the JVM. Oracle welcomes suggestions based upon your experiences with migrating applications to the Oracle JRockit JDK.

Migrating an application to the JRockit JDK requires you to make a few minor environmental changes and follow some simple coding guidelines.

This chapter provides instructions and tips to help you complete the migration process. It also describes the benefits of migration and problems that you might encounter during migration, and it describes some best Java coding practices to ensure that your application runs successfully after migrating it to the JRockit JDK.

The JRockit JDK is the default JDK shipped with Oracle WebLogic Server. Although there are other JDKs available in the market, Oracle recommends that you use JRockit JDK with Oracle products.

You can migrate applications for all platforms for which Oracle WebLogic Server is supported with the JRockit JDK. For more information, see the *Oracle JRockit JDK Supported Configurations* at `http://www.oracle.com/technology/software/products/ias/files/fusion_certification.html`.

5.1 Support for Migration Problems

If you have any problems while migrating an application to the JRockit JDK, contact Oracle through My Oracle Support.

When you contact Oracle for support, provide as much information as possible about the problem; for example, provide the following information:

- Hardware
- Operating system and its version
- Program you are attempting to migrate
- Stack dumps (if any)
- A small code example that reproduces the error
- Copies of any `*.dump` and `*.mdmp/core` process memory dump files (on Windows, the dump files they are stored as `*.mdmp`; on Linux and Solaris as `core` or `core.*`)

5.2 Migration Tasks

The following topics describe the tasks you must perform to migrate applications from a third-party JDK to the Oracle JRockit JDK.

- Section 5.2.1, "Make Necessary Changes in the Application Environment"
- Section 5.2.2, "Tune the JRockit JVM for Optimal Performance"

5.2.1 Make Necessary Changes in the Application Environment

Note: The changes described in this section apply primarily to Oracle WebLogic Server. If you are working with other Java applications, you must change the scripts and environments according to how that application is set up.

To migrate from a third-party JDK to the JRockit JDK, make the following changes to the application environment.

- Set the `JAVA_HOME` environment variable in *`WL_HOME`*`/common/commEnv.cmd` (or `.sh`) to the appropriate path.
- Set the `JAVA_VENDOR` environment variable in *`WL_HOME`*`/common/commEnv.cmd` (or `.sh`) to `BEA`.
- If you use a start-up script, remove any command-line options that are specific to third-party JVMs. If possible, replace the third-party options with the Oracle JRockit JVM-specific equivalents (for example, `-Xns`). Other flags that might need to be changed include `MEM_ARGS` and `JAVA_VM`.

  For more information about the command-line options that the Oracle JRockit JVM supports, see the *Oracle JRockit Command-Line Reference*:
- Change `config.xml` to point the default compiler settings to the `javac` compiler in the JRockit JDK.

Note: Ensure that your application is developed to run on the Oracle JRockit JDK according to the guidelines described in the *Oracle JRockit Application Development Guide*.

5.2.2 Tune the JRockit JVM for Optimal Performance

After you migrate your application to the JRockit JDK, tune the JVM for optimal performance. For example, you might want to specify a different startup heap size or

set custom garbage collection parameters. For more information about tuning the JRockit JVM, see *Oracle JRockit Performance Tuning Guide*.

The nonstandard command-line options (preceded with `-X`) are critical tools for tuning a JVM at startup. These options change the behavior of the JRockit JVM to suit the requirements of different Java applications. If you are migrating an application to the JRockit JDK, Oracle recommends that you become familiar with the nonstandard options available to you. For more information, see the *Oracle JRockit Command-Line Reference*.

Note: The names of nonstandard options might not be the same from JVM to JVM. For example, in the JRockit JVM the name of the nonstandard option to set the nursery in generational concurrent and generational parallel garbage collectors is `-Xns`, whereas, in the HotSpot JVM, the name of the equivalent option is `-XX:NewSize`.

Nonstandard options are subject to change at any time.

5.3 Testing the Application

After performing the migration tasks described earlier, test your application on the JRockit JVM.

Some important reasons for testing are:

- You might find bugs in your application that do not occur on the HotSpot JVM; for example, synchronization problems.
- You might have used third-party class libraries that do not follow the Java specifications and rely on HotSpot-specific classes or behavior.
- You might have used third-party class files that are not correct. The JRockit JVM enforces verification rigorously.

To test your application on the JRockit JVM, run the application against any test scripts or benchmarks that are appropriate for that application. If any problems occur, handle them as you typically would for the specific application.

5.4 Replicating Tools Supplied with the HotSpot JDK

The following J2SE tools, normally available with the HotSpot JDK, are not shipped with the JRockit JDK:

- `jinfo`
- `jhat`
- `jmap`
- `jsadebugd`
- `jstack`

The JRockit JDK provides internal tools that are equivalent to most of the HotSpot tools. Table 5–1 lists the HotSpot JDK tools and their JRockit JDK equivalents. Some tools require using the `jrcmd` feature. For more information, see *Oracle JRockit JDK Tools*.

Table 5–1 JRockit JDK and HotSpot JDK: Tool Equivalents

HotSpot Tool	Shipped with JRockit JDK	Oracle JRockit Equivalent
`jinfo`	No	`jrcmd` *`pid`* `print_properties` `jrcmd` *`pid`* `command_line`
`jmap`	No	JRockit Memory Leak Detector (see *Oracle JRockit JDK Tools* The following diagnostic commands: ▪ `jrcmd` *`pid`* `print_object_summary` ▪ `jrcmd` *`pid`* `verbose_referents` ▪ `jrcmd` *`pid`* `heap_diagnostics`
`jsadebugd`	No	None
`jhat`	Yes	`jhat` works with the JRockit JVM. You can also use the JRockit Memory Leak Detector and the JRockit Flight Recorder.
`jstack`	No	`jrcmd` *`pid`* `print_threads`
`jps`	Yes	`jrcmd`
`jstat`	Yes	`jstat` works with the JRockit JVM. You can also use the JRockit Flight Recorder and the JRockit Management Console in Oracle JRockit Mission Control (see *Oracle JRockit JDK Tools*).
`jstatd`	Yes	`jstatd` works with the JRockit JVM.
`jconsole`	Yes	`jconsole` works with the JRockit JVM. You can also use the JRockit Management Console in Oracle JRockit Mission Control (see *Oracle JRockit JDK Tools*).
`jrunscript`	Yes	`jrunscript` works with the JRockit JVM.

5.5 Command-line Option Compatibility Between the JRockit and HotSpot JVMs

This section describes the compatibility between the important and commonly used command-line options available when running the JRockit JVM with the options when running the HotSpot JVM.

Table 5–2 lists the options that have the same name in both the HotSpot and JRockit JVMs and but have different functions.

Table 5–2 Same Name, Different Function

Option	HotSpot JVM Function	JRockit JVM Function
`-Xms`	Sets the initial size of the heap	Sets the initial and minimum size of the heap For more information, see the *Oracle JRockit Command-Line Reference*.

Table 5–3 lists the options that perform the same or similar functions on both the HotSpot and JRockit JVMs but have different names.

Table 5–3 Different Name; Same or Similar Function

HotSpot JVM Option	JRockit JVM Option	Function
`-verbose:gc`	`-Xverbose:gc`	This option prints log information about the memory system.
`-Xmn`, `-XX:NewSize`, and `-XX:MaxNewSize`	`-Xns`	This option sets the size of the young generation.
`-XX:+UseConcMarkSweepGC`	`-Xgc:singlecon`	This option sets the garbage collector to use a concurrent strategy.
`-XX:+UseParallelGC`	`-Xgc:parallel`	This option sets the garbage collector to use a parallel strategy.

Table 5–4 lists options available only when using the Oracle JRockit JVM.

Table 5–4 Options Available Only in the JRockit JVM

Option	Function
`-XpauseTarget`	Specifies a suitable pause time for the application

Note: `-Xint` and `-XX:MaxPermSize`, which are commonly used with the Hotspot JVM, are not valid for the JRockit JVM because the JRockit JVM has neither an interpreter nor a permgen space.

6

Setting Up and Running the Oracle JRockit JDK

Before using the Oracle JRockit JDK, ensure that you set up the environment correctly.

The process of setting up the Oracle JRockit JDK environment involves the following tasks:

- Section 6.1, "Installing the Oracle JRockit JDK"
- Section 6.2, "Setting Up and Verifying Your Windows Environment"
- Section 6.3, "Setting Up and Verifying Your Linux Environment"
- Section 6.4, "Setting Up and Verifying Your Solaris Environment"

6.1 Installing the Oracle JRockit JDK

The JRockit JDK is included in several Oracle products, for example Oracle JRockit Mission Control, Oracle JRockit Real Time and Oracle WebLogic Server. So the JRockit JDK is installed when you install those products. For more information, see the installation guide for your specific Oracle product.

6.2 Setting Up and Verifying Your Windows Environment

Several environment variables control the operation of the JRockit JVM on Windows. For these environment variables to work, the `PATH` environment variable must point to the directory of your Java installation:

```
set PATH=java_installation_directory\bin;%PATH%
```

6.3 Setting Up and Verifying Your Linux Environment

Keep your Linux environment up to date and ensure that you have a release supported by Oracle when running the JRockit JDK.

For the list of Linux releases against which the JRockit JDK has been tested, see *Oracle JRockit JDK Supported Configurations* at http://www.oracle.com/technology/software/products/ias/files/fusion_certification.html.

The following is the correct path for Linux installations:

```
export PATH=java_installation_directory/bin:$PATH
```

Verify Whether the glibc Library is Installed for Linux on IA32

When running the JRockit JVM on Linux IA32, the JVM must be configured to use the `glibc` library compiled for the i686 architecture; otherwise, the JRockit JVM might freeze or crash.

Verify which `glibc` library is installed by running the following command:

```
rpm -q --queryformat '\n%{NAME} %{VERSION} %{RELEASE} %{ARCH}\n' glibc
```

Verify That the /proc File System is Mounted When Running in a chroot(3) Environment

In some Linux versions, the `/proc` file system is not mounted when running in a `chroot(3)` environment. This might cause the JRockit JVM to become unstable or crash, because the JVM and some Linux libraries must access information about the hardware platform from the `/proc` file system.

To verify that the `/proc` file system is mounted, run the shell command `getconf _NPROCESSORS_CONF` from the command line in your `chroot(3)` environment. If the command does not return the correct number of processors in your system, you must mount the `/proc` file system before running the JRockit JVM.

6.4 Setting Up and Verifying Your Solaris Environment

The following is the correct path for Solaris installations:

```
export PATH=java_installation_directory/bin:$PATH
```

Note that Oracle JRockit JDK R28.x is supported for Solaris only on 64-bit SPARC systems.