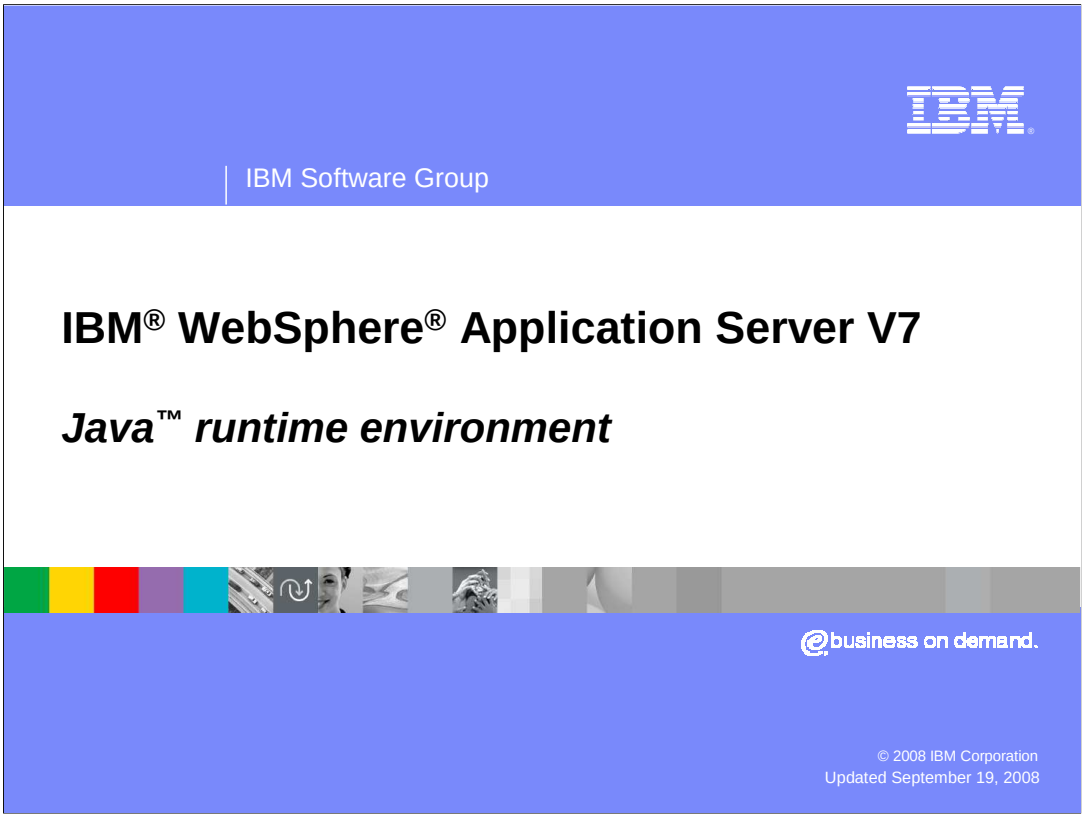


This presentation describes the components of the IBM runtime environment for Java.

Agenda

- Overview
- Shared data
- Just-in-time compiler
- Memory management

The first section of the presentation provides a brief overview of Java runtime environments that are available from IBM and supported with WebSphere Application Server V7. The rest of the presentation focuses on the components that make up the IBM runtime environment for Java, including the ability to share data among Java Virtual Machines, new features in the just-in-time compiler, and updates in memory management and garbage collection.

Section

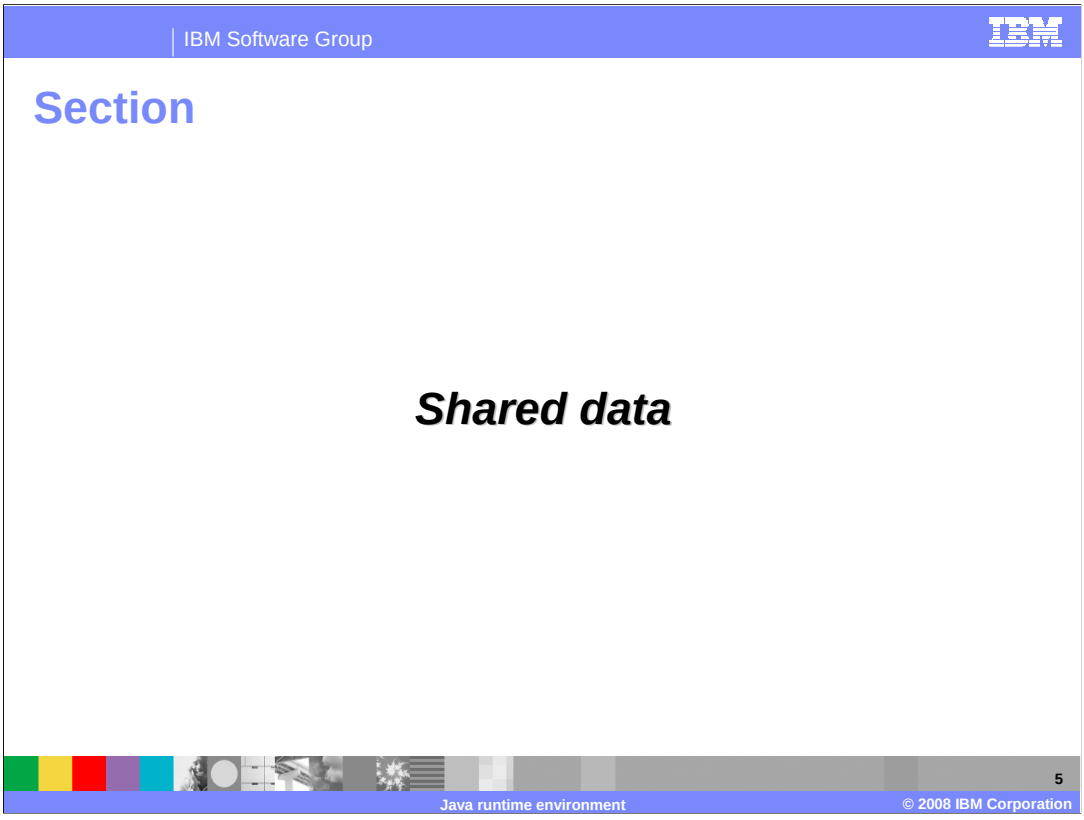
Overview

This section describes the Java runtimes that are available with WebSphere Application Server V7.

Runtime environment

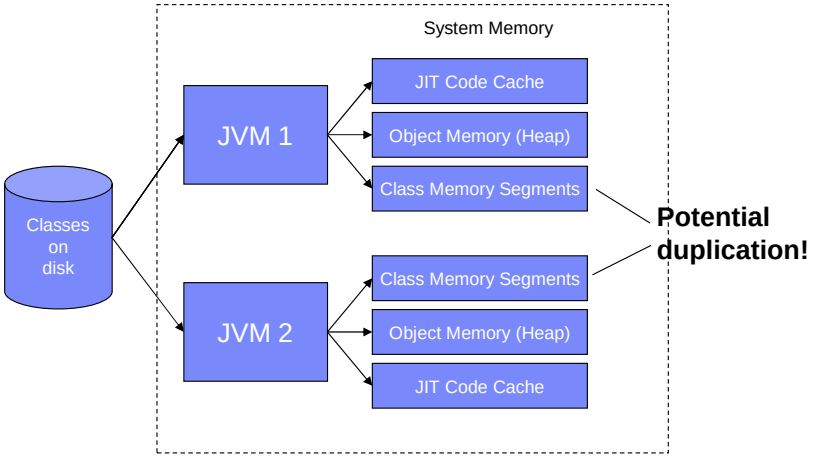
- On Windows®, AIX®, Linux®, z/OS®
  - The runtime is built on the underlying technology from Java 5
  - Enhancements in the virtual machine, garbage collector, just-in-time compiler
- On the i5/OS® platform, the Java runtime is a part of the operating system
  - Not bundled with the application server
  - Two versions are available: J9 and Classic
- Solaris and HP platforms use a hybrid JDK
  - Runtime components do not come from IBM
  - Bundled with IBM XML, security, and ORB library

Java runtime support for WebSphere Application Server varies by platform. The runtime environment on Windows, AIX, Linux, and z/OS is an IBM implementation of the Java runtime, based on the underlying technology in the IBM SDK for Java 5. In Version 6, there are several enhancements in the virtual machine, including memory management and the just-in-time compiler. The rest of this presentation will focus on the updates in the IBM SDK for Java Version 6. On all platforms except i5/OS, the Java runtime comes packaged as a part of the WebSphere Application Server product. On the System i, the Java runtime is a part of the operating system. There are two versions of the JDK available for WebSphere Application Server V7 on i5/OS. On this platform, users need to have a supported Java product installed before installing the application server. On Solaris and HP, the application server is bundled with a hybrid JDK. This JDK contains runtime components based on the Java 6 reference implementation, bundled with class libraries from the IBM implementation. On all platforms, the supported Java runtime environments for WebSphere Application Server V7 are compliant with the Java SE Platform Version 6 specification.



The shared data cache allows Java Virtual Machines running on one system to share data with each other. The next section provides information on the motivation for the shared cache and enhancements to the cache in the Java 6 runtime.

Shared cache motivation



In the Java Virtual Machine, Java classes get loaded from disk into system memory. There is a lot of information related to each class that ends up getting stored in memory – including static class data, methods that have been compiled by the just-in-time compiler, and objects that have been instantiated based on the class definition. In a typical system, each JVM references its own portion of system memory that contains all three elements, like in the diagram on this slide. Some of the information stored in the class memory segments, though, is static class data. Since the data is static, that means that each JVM on a particular system that has loaded that class is duplicating a small amount of information in its associated memory space. Say that you have four JVMs running on one system, and they have all loaded the class java.lang.String. Then the static data associated with the String class is taking up space in system memory four times – once for each Java Virtual Machine that is using that class. If the JVM can identify this duplication and pull out that common data and store it in one place, then system memory consumption can be reduced. This is the idea behind creating a shared data cache.

Shared data cache overview

- The shared cache was introduced in IBM's SDK for Java 5
- Static class data was stored in an area of memory accessible by any Java Virtual Machine (JVM) on the system
- Cache was persistent beyond the lifetime of any JVM but was lost on system shutdown
- Offered significant startup speed improvements and memory savings

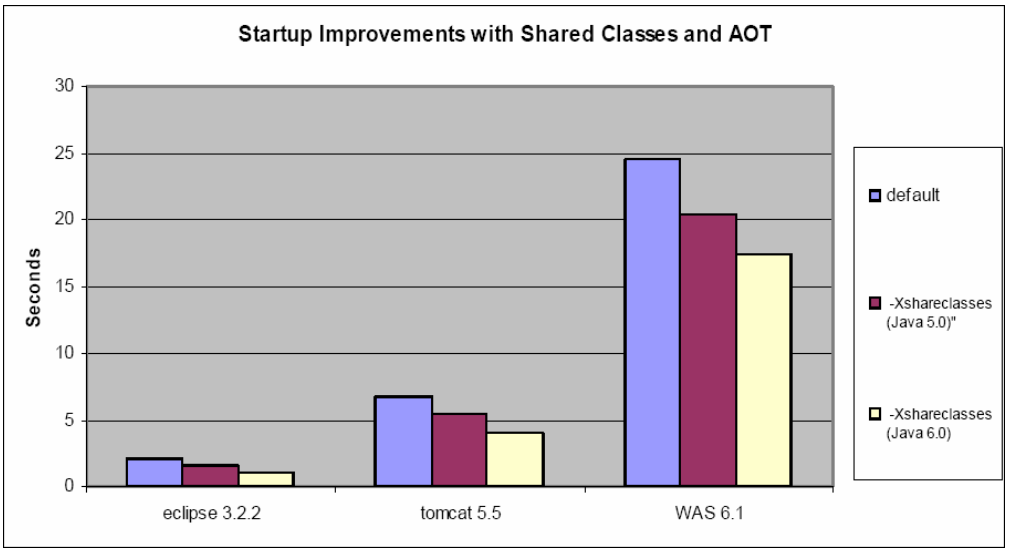
The shared cache was introduced in the IBM SDK for Java Version 5, to provide a mechanism for pulling out duplicated class data and storing it in a single location. Static class data can be stored in an area of shared memory that is accessible by any Java Virtual Machine that is running on the system. In Java 5, the cache can be persisted beyond the lifetime of any JVM, but since it resided only in memory, the cache was lost on system shutdown. Enabling the shared class cache has two major advantages. First, static class data was no longer duplicated across JVMs, so overall memory usage is reduced. The more JVMs you have running on a system that are connecting to the same shared cache, the greater the memory savings. Second, when class data is already resident in memory, it no longer has to be spooled in from disk when a class is loaded. Loading information from memory is much faster than pulling it in from a hard drive. The time period that a JVM is starting up is a period of intense class loading. Enabling the shared cache will reduce this class loading time and cause an improvement in JVM startup speed.

Shared data cache updates

- Shared cache persistence
  - Cache information can be written to a memory-mapped file
  - Initial JVM startup after a reboot no longer takes a performance hit
  - You can control persistence using command line options
    - `-Xshareclasses:persistent, -Xshareclasses:nonpersistent`
- Pre-compilation of Java code
  - Store ahead of time (AOT) compiled code generated by the just-in-time (JIT) compiler
  - Automatically update methods if they are re-compiled
  - Improves JVM startup time
  - **Example:** `-Xscmx50M -Xscminaot5M -Xscmaxaot10M`
    - Creates a 50M cache, guaranteeing at least 5M of space but no more than 10M for AOT compiled code

In the Java 6 runtime, the shared cache has been improved. Now, the shared cache can be written to a memory-mapped file so that it will persist beyond system reboot. In the previous release, the first time that a JVM was started after system reboot, the start up performance was slower because the shared cache needed to be repopulated. In this release, you now have the ability to make the cache persistent, so it can always be populated (unless you clear the cache manually). Cache persistence can be controlled with command-line options. The types of data that can be stored in the cache have also changed in Java 6. In addition to storing static class data in the cache, you can now also store ahead of time compiled methods. These methods are compiled by the just-in-time compiler and are automatically kept up-to-date in the cache if recompiled at a higher optimization level. The compiler heuristically identifies methods that are most heavily used in JVM startup and focuses on caching these in the shared data area to achieve the maximum startup speed improvements. If data sharing is enabled, then ahead of time compiled methods are cached by default. There are command-line settings that allow you to control how much of the cache is allocated to compiled code.

Shared data cache performance



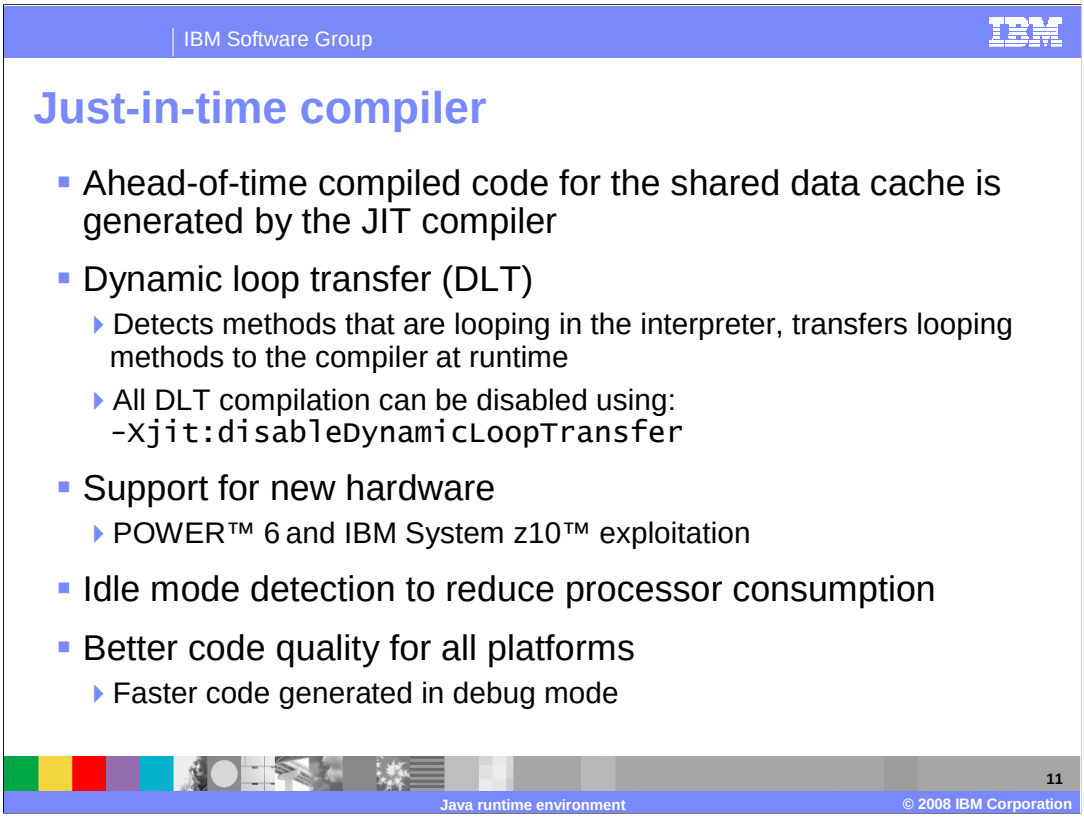
Lower is better!

This graph illustrates the startup speed improvement provided by the shared data cache for different applications. The improvements in the Java 5 level are a result of not having to load all of the class data in from disk, and the improvements in Java 6 are a result of not having to compile all of the class methods because of the ahead of time compiled code already stored in the cache. Consider the WebSphere Application Server column on the far right of the graph. Enabling the shared data technology from Java 5, which includes static class data only, results in approximately 18% faster start up times than the default settings. When the Java 6 shared cache, which includes compiled code, is enabled, start up speed improves by around 30%, compared with the default settings.

Section

Just-in-time compiler

This section describes updates in the just-in-time compiler.



Enhancements in the just-in-time compiler are focused on improved performance and support for new hardware. As discussed in the previous section, the compiler is responsible for compiling the ahead of time methods that are stored in the shared data cache. This new feature can greatly improve the startup speed of a Java Virtual Machine. The compiler also contains a new dynamic loop transfer capability. Java code runs both interpreted and compiled. Dynamic loop transfer supports breaking into the interpreter and transferring methods that are running long loops over into the compiler. The compiler can then optimize the method so that it runs much faster. The just-in-time compiler supports the latest IBM processor technology, including POWER 6 and the specialized condensed instructions available on the z10. The compiler has also been optimized to reduce idle mode processor consumption and generate better quality code for all platforms.

Section

Memory management

This section describes memory management changes in Java 6, including changes in the garbage collector and the new compressed references scheme for 64-bit platforms.

Garbage collector

- Significantly faster class loader load/unload performance and improved footprint
- Changes to gencon GC policy
  - Hierarchical scanning – improve locality and maintain scalability
  - Removed 64mb cap on default new space size
  - Default maximum value of `-Xmns` and `-Xmnx` is now 25% heap

Overall improvements in the garbage collector have been introduced to improve class loader performance and lower memory footprint. Additionally, the generational concurrent, or gencon, garbage collection policy has been enhanced. The policy includes new hierarchical scanning technology, which works to improve the locality of objects in the heap, which improves overall performance and makes the environment more scalable. In the previous, the default maximum nursery size was 64 MB. In some environments with large heaps, say several gigabytes of heap space, a 64 MB nursery is too small and causes the nursery space and the tenured object space to be out of balance, which can negatively impact performance. To make the initial nursery settings more friendly to large heaps, it no longer defaults to 64 MB, but is set to 25% of the total heap size.

Compressed references

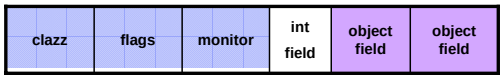
- Available in 64-bit IBM SDKs running with limited heap
- Use 32-bit heap offsets to reference heap data
- Allows larger heap sizes with increased efficiency
- Disabled by default, enable with `-Xcompressedrefs`

Pointer size	Space	Max heap	Efficiency
31 bits	2 GB	1.3 GB (z/OS)	100%
32 bits	4 GB	1.7 GB (Win) 3.2 GB (AIX)	100%
64 bits compressed	4 to 32 GB	4 to 32 GB	~70-100%
64 bits	16 EB	16 EB	50-70%

Platforms that use 31- and 32-bit address spaces have limited memory space that can be made available to the Java heap. On Windows, for example, the maximum Java heap space that is available with most garbage collection policies is 1.7 GB. If you have an application that you want to run on Windows and it requires a 3 GB heap, then your only option is to switch to a 64-bit version of the operating system. On a 64-bit system, the addressable space available to applications is essentially unlimited, at 16 exabytes, where an exabyte is a quintillion bytes. While 64-bit platforms support larger heap sizes and have increased throughput compared to 32-bit platforms, they also use 64-bit pointers to reference objects. Since the Java heap is made up mostly of object references, and moving from a 32-bit platform to a 64-bit platform causes the pointer size to double from 32-bits to 64-bits, then running an application that needed about 3 GB of memory on a 32-bit platform requires between 5 and 6 GB on a 64-bit platform. Compressed references is a new technology that has been introduced in this release that allows 64-bit JVMs to use 32-bit heap offsets to reference heap data. This allows applications to have access to a larger memory space than 32-bit platforms and substantially reduces the overall memory footprint typically required to run Java on a 64-bit platform.

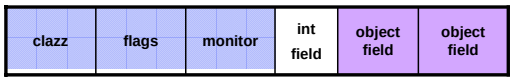
Object structure

- 32-bit Object (24 bytes – 100%)



- 64-bit Object (48 bytes – 50%)

- 64-bit Compressed References (24 bytes –100%)



  - Use 32-bit values (offsets) to represent object fields
  - With scaling, can address between 4GB and 32GB

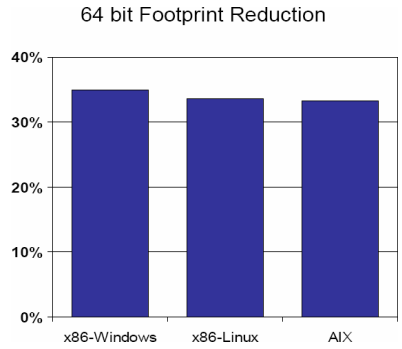
The diagrams on this page illustrate the concept of heap usage efficiency. The benchmark for efficiency is the 32-bit environment, which requires 24 bytes to store object data. This 24 byte measurement is considered 100% efficiency. To store that same object on a 64-bit platform requires 48 bytes of object data. This is twice the size of the space needed on the 32-bit platform, so is only considered 50% efficient. However, in a 64-bit compressed references environment using a 4 GB heap or smaller, the usage efficiency is exactly the same as on a 32-bit platform – 100%. This is because compressed references uses 32-bit offsets to reference the heap, which take up half the space of 64-bit pointers. Four gigabytes is the maximum address space supported by 32-bits, but it is possible for the JVM to use internal scaling to address up to 32 GB of heap with compressed references. As heap grows above 4 GB, efficiency will decrease slightly.

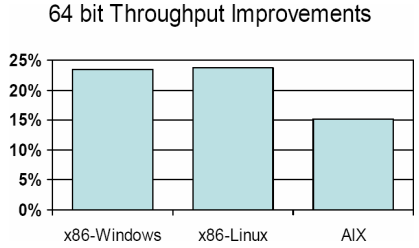
Benefits of compressed references

- Access to larger heap space than 32-bit environments
- Smaller heaps than 64-bit environments
- Overall throughput improvements
- More efficient heap utilization

Get **SMALLER**,

Get **FASTER**





Using compressed references on 64-bit platforms allows the JVM to access larger heap space than 32-bit environments while maintaining a smaller overall heap size than 64-bit environments – this is due to the more efficient heap utilization made possible by using smaller pointers inside Java objects. In addition, applications still receive the throughput benefits of running in a 64-bit environment. Overall, compressed references enable Java applications to run faster with a smaller memory footprint.

Section

Summary and references

This section contains a summary and references.

Summary

- Shared cache allows sharing class data and compiled methods between JVMs
- Just-in-time compiler enhancements focused on improved performance
- Compressed references allow smaller memory usage on 64-bit platforms

The IBM runtime environment for Java 6 includes significant changes from the previous release, all designed to improve Java application performance. The shared data cache allows compiled code to be shared between JVMs, which improves start up time. The just-in-time compiler incorporates several optimizations to improve the quality and performance of compiled code, and compressed references allow smaller heap usage on 64-bit platforms.

References

- Diagnostics Guide
  http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/index.jsp

Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send e-mail feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_WASv7_JavaRuntime.ppt

This module is also available in PDF format at: ../WASv7_JavaRuntime.pdf

You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, copyrights, and disclaimers

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX i5/OS IBM POWER System z System z10 WebSphere z/OS

A current list of other IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

Windows and the Windows logo are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Java runtime environment, JDK, JVM, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements or changes in the products or programs described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

Information is provided "AS IS" without warranty of any kind. THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (for example, IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products.

IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Note to U.S. Government Users - Documentation related to restricted rights-Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract and IBM Corp.