The Shared Classes feature helps reduce memory footprint and improves startup performance

Ben Corrie

May 30, 2006

The new Shared Classes feature in the IBM implementation of version 5.0 of the Java™ platform offers a completely transparent and dynamic means of sharing all loaded classes that places no restrictions on the JVMs that are sharing the class data. This feature offers a straightforward and flexible solution for reducing virtual memory footprint and improving startup time, and there are few applications that will not benefit from it. This article explores how the feature works, how to use it, and when to use it, along with some of the features it provides.

The principle of sharing loaded classes between Java virtual machine (JVM) processes is not new. Sun’s CDS feature, for example, writes system classes into a read-only file that is memory-mapped into the JVM. The Shiraz feature in the IBM z/OS® 1.4.2 JVM used a master JVM to populate a class cache that was then shared by slave JVMs.

The IBM implementation of the 5.0 JVM takes the concept a step further by allowing all system and application classes to be stored in a persistent dynamic class cache in shared memory. This Shared Classes feature is supported on all of the platforms on which the IBM implementation of the JVM ships. The feature even supports integration with runtime bytecode modification, which this article discusses later.

The Shared Classes feature has been designed from the ground up to be an option you can just switch on and forget about, yet it provides very powerful scope for reducing virtual memory footprint and improving JVM startup time. For this reason, it is best suited to environments where more than one JVM is running similar code or where a JVM is regularly restarted.

In addition to the runtime class-sharing support in the JVM and its classloaders, there is also a public Helper API provided for integrating class sharing support into custom classloaders, which this article discusses in detail.

How it works
Let’s start by exploring the technical details of how the Shared Classes feature operates.
Enabling class sharing

You enable class sharing by adding `-Xshareclasses[:name=<cachename>]` to an existing Java command line. When the JVM starts up, it looks for a class cache of the name given (if no name is provided, it chooses a default name) and it either connects to an existing cache or creates a new one, as required.

You specify cache size using the parameter `-Xscmx<size>[k|m|g]`; this parameter only applies if a new cache is created by the JVM. If this option is omitted, a platform-dependent default value is chosen (typically 16MB). Note that there are operating system settings that can limit the amount of shared memory that can be allocated -- for instance, `SHMMAX` on Linux is typically set to about 20MB. The details of these settings can be found in the Shared Classes section of the appropriate user guide (see the Related topics section for a link).

The class cache

A class cache is an area of shared memory of fixed size that persists beyond the lifetime of any JVM using it. Any number of shared class caches can exist on a system, subject to operating system settings and restrictions; however, a single JVM can only connect to one cache during its lifetime.

No JVM owns the cache, and there is no master/slave JVM concept; instead, any number of JVMs can read and write to the cache concurrently. A cache is deleted either when it is explicitly destroyed using a JVM utility or when the operating system restarts (a cache cannot persist beyond an operating system restart). A cache cannot grow in size and, when it becomes full, a JVM can still load classes from it but cannot add any classes to it. There are a number of JVM utilities to manage active caches, which the section entitled "Shared classes utilities" discusses.

How are classes cached?

When a JVM loads a class, it first looks in the cache to see if the class it needs is already present. If it is, it loads the class from the cache. Otherwise, it loads the class from the filesystem and writes it into the cache as part of the `defineClass()` call. Therefore, a non-shared JVM has the following classloader lookup order:

1. Classloader cache
2. Parent
3. Filesystem
In contrast, a JVM running with Class Sharing uses the following order:

1. Classloader cache
2. Parent
3. Shared cache
4. Filesystem

Classes are read from and written to the cache using the public Helper API, which has been integrated into the IBM implementation of `java.net.URLClassLoader`. Therefore, any classloader that extends `java.net.URLClassLoader` gets class sharing support for free.

**What parts of the class are cached?**

Inside the IBM implementation of the JVM, Java classes are divided into two parts: a read-only part called a ROMClass, which contains all the class's immutable data, and a RAMClass that contains data that is not immutable, such as static class variables. A RAMClass points to data in its ROMClass, but the two are completely separate, which means that it is quite safe for a ROMClass to be shared between JVMs and even between RAMClasses in the same JVM.

In the non-shared case, when the JVM loads a class, it creates the ROMClass and the RAMClass separately and stores them both in its local process memory. In the shared case, if the JVM finds a ROMClass in the class cache, it only needs to create the RAMClass in its local memory; the RAMClass then references the shared ROMClass.

Because the majority of class data is stored in the ROMClass, this is where the virtual memory savings are made. (The "Virtual memory footprint" sections discusses this in more detail.) JVM startup times are also significantly improved with a populated cache because some of the work to define each cached class has already been done and the classes are being loaded from memory, rather than from the filesystem. Startup time overhead to populate a new cache (which I discuss later in this article) is not significant, as each class simply needs to be relocated into the cache as it is defined.

**What happens if a class changes on the filesystem?**

Because the cache can persist indefinitely, filesystem updates that invalidate classes in the cache may occur. It is therefore the responsibility of the cache code to ensure that, if a classloader makes a request for a shared class, then the class returned should always be exactly the same as the one that would have been loaded from the filesystem. This happens transparently when classes are loaded, so users can modify and update as many classes as they like during the lifetime of a shared class cache, knowing that the correct classes are always loaded.

**Pitfalls with class changes: Two examples**

Imagine a class C1 that is loaded into the cache by a JVM. Then, when the JVM shuts down, C1 is changed and recompiled. When the JVM restarts, it definitely should not load the cached version of C1.

Similarly, imagine a JVM that's running with a classpath of `/mystuff:/mystuff/myClasses.jar`. It loads C2 from myClasses.jar into the cache. Then a different C2.class
The JVM detects filesystem updates by storing timestamp values into the cache and comparing the cached values with actual values on each class load. If it detects that a JAR file has been updated, it has no idea which classes have been changed, so all classes loaded from that JAR in the cache are immediately marked as stale and cannot then be loaded from the cache. When the classes from that JAR are loaded from the filesystem and re-added to the cache, only the ones that have actually changed are added in their entirety; those that haven’t changed are effectively made not stale.

Classes cannot be purged from the cache, but the JVM attempts to make the most efficient use of the space it has. For example, the same class is never added twice, even if it is loaded from many different locations. So, if the same class C3 is loaded from /A.jar, /B.jar, and /C.jar by three different JVMs, the class data is only added once, but there are three pieces of metadata to describe the three locations from which it was loaded.

**Shared classes utilities**

There are a number of utilities that can be used to manage active caches, all of which are suboptions to `-Xshareclasses`. (You can get a complete list of all valid suboptions to `-Xshareclasses` by typing `java -Xshareclasses:help`.)

Note that none of the utilities (apart from `expire`) actually start the JVM, so they perform the action required and then exit without running a class. Note also that the message `Could not create the Java virtual machine` is printed by the Java launcher by each utility because a JVM is not started. This is not an error.

To demonstrate the use of these options, let’s walk through some examples. First, let’s create two caches by running a `HelloWorld` class with different cache names, as shown in Listing 1:

### Listing 1. Creating two caches

```sh
c:\j9vmwi3223\sdk\jre\bin>java -cp . -Xshareclasses:name=cache1 HelloWorld
Hello
C:\j9vmwi3223\sdk\jre\bin>java -cp . -Xshareclasses:name=cache2 HelloWorld
Hello
```

Running the `listAllCaches` suboption lists all caches on a system and determines whether or not they are in use, as in you can see in Listing 2:

### Listing 2. Listing all caches

```sh
c:\j9vmwi3223\sdk\jre\bin>java -Xshareclasses:listAllCaches
Shared Cache   Last detach time
cache1         Sat Apr 15 18:47:46 2006

Could not create the Java virtual machine.
```
Running the `printStats` option prints summary statistics on the named cache, as shown in Listing 3. For a detailed description of what all the fields shown here mean, consult the user guide (see Related topics for a link).

**Listing 3. Summary statistics for a cache**

```bash
C:\j9vmw13223\sdk\jre\bin>java -Xshareclasses:name=cache1,printStats

Current statistics for cache "cache1":

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>base address</td>
<td>0x41D10058</td>
</tr>
<tr>
<td>end address</td>
<td>0x42D0FFF8</td>
</tr>
<tr>
<td>allocation pointer</td>
<td>0x41E3B948</td>
</tr>
<tr>
<td>cache size</td>
<td>16777128</td>
</tr>
<tr>
<td>free bytes</td>
<td>15536080</td>
</tr>
<tr>
<td>ROMClass bytes</td>
<td>1226992</td>
</tr>
<tr>
<td>Metadata bytes</td>
<td>14056</td>
</tr>
<tr>
<td>Metadata % used</td>
<td>1%</td>
</tr>
<tr>
<td># ROMClasses</td>
<td>313</td>
</tr>
<tr>
<td># Classpaths</td>
<td>2</td>
</tr>
<tr>
<td># URLs</td>
<td>0</td>
</tr>
<tr>
<td># Tokens</td>
<td>0</td>
</tr>
<tr>
<td># Stale classes</td>
<td>0</td>
</tr>
<tr>
<td>% Stale classes</td>
<td>0%</td>
</tr>
</tbody>
</table>

Cache is 7% full

Could not create the Java virtual machine.
```

Running the `printAllStats` option on a named cache lists the entire contents of the cache, along with the `printStats` summary information. Each class stored in the cache is listed along with the context data, such as classpath data. In Listing 4, you can see the JVM’s bootstrap classpath listed, followed by some of the classes and details of where they were loaded from:

**Listing 4. Listing the entire contents of a cache**

```bash
C:\j9vmw13223\sdk\jre\bin>java -Xshareclasses:name=cache1,printAllStats

Current statistics for cache "cache1":

1: 0x42D0FAB0 CLASSPATH
   C:\j9vmw13223\sdk\jre\lib\vm.jar
   C:\j9vmw13223\sdk\jre\lib\core.jar
   C:\j9vmw13223\sdk\jre\lib\charsets.jar
   C:\j9vmw13223\sdk\jre\lib\graphics.jar
   C:\j9vmw13223\sdk\jre\lib\security.jar
   C:\j9vmw13223\sdk\jre\lib\ibmpkcs.jar
   C:\j9vmw13223\sdk\jre\lib\ibmorb.jar
   C:\j9vmw13223\sdk\jre\lib\ibmcfw.jar
   C:\j9vmw13223\sdk\jre\lib\ibmorbapi.jar
   C:\j9vmw13223\sdk\jre\lib\ibmjcefw.jar
   C:\j9vmw13223\sdk\jre\lib\ibmgssprovider.jar
   C:\j9vmw13223\sdk\jre\lib\ibmjseprovider2.jar
   C:\j9vmw13223\sdk\jre\lib\ibmjaaslm.jar
   C:\j9vmw13223\sdk\jre\lib\ibmjaasactivelm.jar
   C:\j9vmw13223\sdk\jre\lib\ibmcertpathprovider.jar
   C:\j9vmw13223\sdk\jre\lib\server.jar
   C:\j9vmw13223\sdk\jre\lib\xml.jar
1: 0x42D0FAB0 ROMCLASS: java/lang/Object at 0x41D10058.
   Index 0 in classpath 0x42D0FAB0.
1: 0x42D0FA50 ROMCLASS: java/lang/J9VMInternals at 0x41D106E0.
```

Java technology, IBM style, Part 4: Class sharing
Named caches are destroyed using the `destroy` option, illustrated in Listing 5. Similarly, `destroyAll` destroys all caches that are not in use and that the user has permissions to destroy.

**Listing 5. Destroying a cache**

```
C:\j9vmw\i3223\sdk\jre\bin>java -Xshareclasses:name=cache1,destroy
JVMSHRC010I Shared Cache "cache1" is destroyed
Could not create the Java virtual machine.

C:\j9vmw\i3223\sdk\jre\bin>java -Xshareclasses:listAllCaches
Shared Cache Last detach time
cache2 Sat Apr 15 18:51:15 2006
Could not create the Java virtual machine.
```

The `expire` option, illustrated in Listing 6, is a housekeeping option that you can add to the command line to automatically destroy caches to which nothing has been attached for a specified number of minutes. This is the only utility that does not cause the JVM to exit. Listing 6 looks for caches that have not been used for a week and destroys them before starting the VM:

**Listing 6. Destroying caches that haven't been used in a week**

```
C:\j9vmw\i3223\sdk\jre\bin>java -cp . -Xshareclasses:expire=10000,name=cache1 Hello
Hello
```

**Verbose options**

Verbose options provide useful feedback on what class sharing is doing. They are all suboptions to `-Xshareclasses`. This section offers some examples of how to use verbose output.

The `verbose` option, illustrated in Listing 7, gives concise status information on JVM startup and shutdown:

**Listing 7. Getting JVM status information**

```
C:\j9vmw\i3223\sdk\jre\bin>java -cp . -Xshareclasses:name=cache1,verbose Hello
[-Xshareclasses verbose output enabled]
JVMSHRC158I Successfully created shared class cache "cache1"
JVMSHRC166I Attached to cache "cache1", size=1677712 bytes
Hello
JVMSHRC168I Total shared class bytes read=0. Total bytes stored=1176392
```

The `verboseIO` option prints a status line for every class load request to the cache. To understand `verboseIO` output, you should understand the classloader hierarchy, as this can be clearly seen for classes that are loaded by any non-bootstrap classloader. Every classloader must delegate up the hierarchy to the bootstrap loader to find a class. In the output, each classloader is assigned a unique ID, but the bootstrap loader is always 0.

Note that it is normal for `verboseIO` to sometimes show classes being loaded from disk and stored in the cache even if they are already cached. For example, the first class load from each JAR on
the application classpath is always loaded from disk and stored, regardless of whether it exists in the cache or not.

In **Listing 8**, the first section demonstrates the population of the cache and the second shows the reading of the cached classes:

The `verboseHelper` suboption, illustrated in **Listing 9**, is an advanced option that gives status output from the Helper API. It is designed to help developers using the Helper API understand how it is being driven. More details on this output are described in the JVM diagnostics guide (see **Related topics** for a link).

### Runtime bytecode modification

Runtime bytecode modification is becoming a popular means of instrumenting behaviour into Java classes. It can be performed using the JVM Tools Interface (JVMTI) hooks (see **Related topics** for a link); alternately, the class bytes can be replaced by the classloader before the class is defined. This presents an extra challenge to class sharing, as one JVM may cache instrumented bytecode that should not be loaded by another JVM sharing the same cache.

However, because of the dynamic nature of the IBM Shared Classes implementation, multiple JVMs using different types of modification can safely share the same cache. Indeed, if the bytecode modification is expensive, caching the modified classes has an even greater benefit, as the transformation only ever needs to be performed once. The only proviso is that the bytecode modifications should be deterministic and predictable. Once a class has been modified and cached, it cannot then be changed further.

Modified bytecode can be shared by using the `modified=<context>` suboption to `-Xshareclasses`. The context is a user-defined name that creates a partition in the cache into which all of the classes loaded by that JVM are stored. All JVMs using that particular modification should use the same modification context name, and they all load classes from the same cache partition. Any JVM using the same cache without the `modified` suboption finds and stores vanilla classes as normal.

### Potential pitfalls

If a JVM is running with a JVMTI agent that has registered to modify class bytes and the `modified` suboption is *not* used, class sharing with other vanilla JVMs or with JVMs using other agents is still managed safely, albeit with a small performance cost because of extra checking. Thus, it is always more efficient to use the `modified` suboption.

Note that this is only possible because the JVM knows that bytecode modification is imminent because of the presence of the JVMTI agent. Therefore, if a custom classloader modifies class bytes before defining the class without using JVMTI *and* without using the `modified` suboption, the classes being defined are assumed to be vanilla and could be incorrectly loaded by other JVMs.

For more detailed information on sharing modified bytecode, see the JVM diagnostics guide (see **Related topics**).
Using the Helper API

The Shared Classes Helper API has been provided by IBM so that developers can integrate class sharing support into custom classloaders. This is only required for classloaders that do not extend java.net.URLClassLoader, as those classloaders automatically inherit class-sharing support.

A comprehensive tutorial on the Helper API is beyond the scope of this article, but here is a general overview. If you'd like a more detailed description, the full Javadoc is available in the Download section, and the diagnostics guide (see Related topics) also has more information.

OSGi support

A good example of a custom classloader framework is OSGi, which is used in Eclipse 3.x. Unless class sharing support is added to OSGi plug-in classloaders, the only classes that will be shared are the bootstrap classes and the application classes needed to start the Eclipse and OSGi frameworks. None of the plug-in code can be shared, as this is loaded by custom OSGi plug-in classloaders, which don't have class sharing support. It is worth noting that in this specific case, IBM provides an OSGi Adaptor plug-in to enable class sharing support in plug-in classloaders. This is available in the file com.ibm.cds_1.0.0.zip, which you can obtain from the Download section.

The Helper API: A summary

All the Helper API classes are in the com.ibm.oti.shared package and are contained within vm.jar in the jre/lib directory. Each classloader wishing to share classes must get a SharedClassHelper object from a SharedClassHelperFactory. The SharedClassHelper once created belongs to the classloader that requested it and can only store classes defined by that classloader. The SharedClassHelper gives the classloader a simple API for finding and storing classes in the class cache to which the JVM is connected. If the classloader is garbage collected, its SharedClassHelper is also garbage collected.

Using the SharedClassHelperFactory

The SharedClassHelperFactory is a singleton that is obtained using the static method com.ibm.oti.shared.Shared.getSharedClassHelperFactory(), which returns a factory if class sharing is enabled in the JVM; otherwise, it returns null.

Using the SharedClassHelpers

There are three different types of SharedClassHelper that can be returned by the factory, each of which is designed for use by a different type of classloader:

- **SharedClassURLClasspathHelper**: This helper is designed for use by classloaders that have the concept of a URL classpath. Classes are stored and found in the cache using the URL classpath array. The URL resources in the classpath must be accessible on the filesystem for the classes to be cached. This helper also carries some restrictions on how the classpath can be modified during the lifetime of the helper.
- **SharedClassURLHelper**: This helper is designed for use by classloaders that don't have the concept of a classpath and can load classes from any URL. The URL resources given must be accessible on the filesystem for the classes to be cached.
- **SharedClassTokenHelper**: This helper effectively turns the shared class cache into a simple hashtable -- classes are stored against string key tokens that are meaningless to the cache.
This is the only helper that doesn’t provide dynamic update capability because the classes stored have no filesystem context associated with them.

Each `SharedClassHelper` has two basic methods, the parameters of which differ slightly between helper types:

- `byte[] findSharedClass(String classname...)` should be called after the classloader has asked its parent for the class (if one exists). If `findSharedClass()` does not return `null`, the classloader should call `defineClass()` on the byte array returned. Note that this function returns a special cookie for `defineClass()`, not actual class bytes, so the bytes cannot be instrumented.

- `boolean storeSharedClass(Class clazz...)` should be called immediately after a class has been defined. The method returns `true` if the class was successfully stored and `false` otherwise.

**Other considerations**

When deploying class sharing with your application, you need to take into account such factors as security, and cache tuning. These considerations are briefly summarised here.

**Security**

By default, class caches are created with user-level security, so only the user that created the cache can access it. For this reason, the default cache name is different for each user so that clashes are avoided. On UNIX, there is a suboption to specify `groupAccess`, which gives access to all users in the primary group of the user that created the cache. However, regardless of the access level used, a cache can only be destroyed by the user that created it or by the root user.

In addition to this, if there is a `SecurityManager` installed, a classloader can only share classes if it has been explicitly granted the correct permissions. Refer to the user guide (see Related topics) for more details on setting these permissions.

**Garbage collection and just-in-time compilation**

Running with class sharing enabled has no effect on class garbage collection (GC). Classes and classloaders can still be garbage collected in just as they are in the non-shared case. Also, there are no restrictions placed on GC modes or configurations when using class sharing.

It is not possible to cache just-in-time (JIT) compiled code in the class cache, so there is no change in behaviour in the JIT when running with class sharing enabled.

**Cache size limits**

The current maximum theoretical cache size is 2GB. The cache size is limited by the following factors:

- **Available disk space (Microsoft Windows only)**. A memory-mapped file is created in a directory called `javasharedresources` to store the class data. This directory is created in the user’s `%APPDATA%` directory. The shared cache files are deleted every time you restart Windows.
• **Available system memory (UNIX only).** On UNIX, the cache exists in shared memory and a configuration file is written to /tmp/javasharedresouces by the JVM to allow all JVMs to locate the shared memory areas by name.

• **Available virtual address space.** Because the virtual address space of a process is shared between the shared class cache and the Java heap, increasing the maximum size of the Java heap reduces the size of the shared class cache you can create.

### An example

To practically demonstrate the benefits of class sharing, this section provides a simple graphical demo. The source and binaries are available from the Download section.

The demo app looks for the jre\lib directory and opens each JAR, calling `class.forName()` on every class it finds. This causes about 12,000 classes to be loaded into the JVM. The demo reports on how long it takes the JVM to load the classes. Obviously, this is a slightly contrived example, as the test only does class-loading, but it effectively demonstrates the benefits of class sharing. Let's run the application and see the results.

### Class-loading performance

1. Download shcdemo.jar from the Download section.
2. Run the test a couple of times without class sharing to warm up the system disk cache, using the command in Listing 10:

   **Listing 10. Warming up the disk cache**

   ```
   C:\j9vmwi3223\sdk\jre\bin>java -cp C:\shcdemo.jar ClassLoadStress
   ```

   When the window in Figure 1 appears, press the button. The app will load the classes.

   **Figure 1. Press the button**

   ![Press my button](image)

   Once the classes have loaded, the application reports how many it loaded and how long it took, as shown in Figure 2:

   **Figure 2. Results are in!**

   ![Load Classes](image)

   You'll notice that the application probably gets slightly faster each time you run it; this is because of operating system optimizations.
3. Now run the demo with class sharing enabled, as illustrated in Listing 11. A new cache is created, so this run shows you the time it takes to populate the new cache. You should specify a cache size of about 50MB to ensure that there is enough space for all the classes. Listing 11 shows the command line and some sample output. As Figure 3 illustrates, this run should take slightly longer than the previous ones, as the demo is populating the shared class cache. You can also optionally use `printStats`, as shown in Listing 12, to see the number of classes stored in the shared class cache:

**Figure 3. Cold cache results**

![Image showing loaded classes](image)

**Listing 12. Seeing the number of cached classes**

```console
C:\j9vmwi3223\sdk\jre\bin>java -Xshareclasses:name=demo,printStats
Current statistics for cache "demo":

<table>
<thead>
<tr>
<th>Base address</th>
<th>0x41D10058</th>
</tr>
</thead>
<tbody>
<tr>
<td>End address</td>
<td>0x44F0FFF8</td>
</tr>
<tr>
<td>Allocation pointer</td>
<td>0x44884030</td>
</tr>
<tr>
<td>Cache size</td>
<td>52428712</td>
</tr>
<tr>
<td>Free bytes</td>
<td>6373120</td>
</tr>
<tr>
<td>ROMClass bytes</td>
<td>45563864</td>
</tr>
<tr>
<td>Metadata bytes</td>
<td>491728</td>
</tr>
<tr>
<td>Metadata % used</td>
<td>1%</td>
</tr>
<tr>
<td># ROMClasses</td>
<td>12212</td>
</tr>
<tr>
<td># Classpaths</td>
<td>3</td>
</tr>
<tr>
<td># URLs</td>
<td>0</td>
</tr>
<tr>
<td># Tokens</td>
<td>0</td>
</tr>
<tr>
<td># Stale classes</td>
<td>0</td>
</tr>
<tr>
<td>% Stale classes</td>
<td>0%</td>
</tr>
</tbody>
</table>

Cache is 87% full

Could not create the Java virtual machine.
```

4. Now, start the demo again with exactly the same Java command line. This time, it should read the classes from the shared class cache, as you can see in Listing 13. You can clearly see the significant improvement in class load time. Again, you should see performance improve slightly each time you run the demo because of operating system optimizations. This particular test was done on a single-processor, 1.6 GHz x86-compatible laptop running Windows XP:
There are a number of variations you can experiment with. For example, you can use the \texttt{javaw} command and start multiple demos and trigger them all loading classes together to see the concurrent performance.

In a real-world scenario, the overall JVM startup time benefit that can be gained from using class sharing depends on the number of classes that are loaded by the application: a HelloWorld program will not show much benefit, whereas a large Web server certainly will. However, this example has hopefully demonstrated that experimenting with class sharing is very straightforward, so you can easily test the benefits.

**Virtual memory footprint**

It is also easy to see the virtual memory savings when running the example program in more than one JVM.

Below are two Task Manager snapshots obtained using the same machine as the previous examples. In Figure 5, five instances of the demo have been run to completion without class sharing. In Figure 6, five instances have been run to completion with class sharing enabled, using the same command lines as before:
Figure 5. Five demos with no class sharing

![Windows Task Manager](image1.png)

Figure 6. Five demos with class sharing enabled

![Windows Task Manager](image2.png)

You can clearly see that the commit charge with class sharing enabled is significantly lower. Windows seems to calculate its commit charge by adding the VM sizes together. Because the total...
amount of cached class data being shared is around 45MB, you can see that the memory usage for each JVM is approximately the VM size plus amount of cached class data.

Both examples started with a commit charge of around 295MB. This means that the first example used 422MB, whereas the second one used 244 MB -- a savings of 178MB.

**Conclusion**

The new Shared Classes feature in the IBM implementation of version 5.0 of the Java platform offers a simple and flexible way to reduce virtual memory footprint and improve JVM startup time. In this article, you have seen how to enable the feature, how to use the cache utilities, and how to get quantifiable measurements of the benefits.

The next article in the series will introduce some of the new debugging, monitoring, and profiling tools available in the IBM implementation of the Java platform. It will also show how you can use them to profile and debug Java applications quickly.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample code</td>
<td>j-ibmjava4.zip</td>
<td>116KB</td>
</tr>
</tbody>
</table>
Related topics

- **JVM user guide**: Essential information on using the IBM SDK.
- **JVM diagnostics guide**: For information on debugging the JDK and more advanced discussions of its features.
- **JVMTI (JVM Tool Interface)**: Reference documentation for JVMTI.
- "**Introduction to the IBM Developer Kit**," Chris Bailey (developerWorks, May 2006): The first article in this series gives an overall summary of the features in the JVM.
- **IBM implementations of Java technology**: Download the SDKs for AIX, Linux, and z/OS, among other IBM developer kits for Java technology, from this page.
- Download the **IBM Development Package for Eclipse**: Develop, test, and run your Java applications with this ready-to-run Java development environment.
- **WebSphere Everyplace Micro Environment**: A production-ready runtime environment, tested and certified to meet J2ME specifications.

© Copyright IBM Corporation 2006
Trademark
(www.ibm.com/developerworks/ibm/trademarks/)