Debugging from dumps

Diagnose more than memory leaks with Memory Analyzer

Skill Level: Intermediate

Chris Bailey (baileyc@uk.ibm.com)
Java Support Architect
IBM

Andrew Johnson (andrew_johnson@uk.ibm.com)
Java Tools Developer and Eclipse Memory Analyzer Tool Committer
IBM

Kevin Grigorenko (kevin.grigorenko@us.ibm.com)
Software Engineer, WebSphere Application Server SWAT Team
IBM

15 Mar 2011

Memory Analyzer is a powerful tool for diagnosing memory leaks and footprint problems from the dump of a Java™ process. It can also give you detailed insight into your Java code and enable you to debug some tricky problems from just one dump, without needing to insert diagnostic code. In this article, you'll learn how to generate dumps and use them to examine the state of your application.

Memory Analyzer variants

The IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer brings the diagnostic capabilities of the Eclipse Memory Analyzer Tool (MAT) to the IBM Virtual Machines for Java by extending Eclipse MAT version 1.0 using the IBM Diagnostic Tool Framework for Java (DTFJ). DTFJ enables Java heap analysis using operating-system-level dumps and IBM Portable Heap Dumps. The IBM variant is available as part of the IBM Support Assistant (ISA). A DTFJ plug-in is available for the stand-alone Eclipse MAT. See Resources for download links.

Adding debug statements to your code to write out the fields in an object, or even entire data collections, is a common problem-solving approach. Often you must do this iteratively as you discover that you need more and more information to understand and solve the problem. Although this process can be effective, it can sometimes fail to bear fruit: The volume of debug code can cause the problem to disappear, you might need to add debug to code you don't own, debugging may require you to restart processes, or the overall performance impact of the debug may prevent the application from running.

Memory Analyzer is a cross-platform, open source tool that you can use not only to diagnose memory problems, but also to gain huge insight into the state and behaviour of an entire Java application. By reading in a snapshot dump created by the Java runtime whilst the application is running, Memory Analyzer gives you a way to diagnose tricky problems that debug code might fail to expose.

This article shows how to generate dumps and use them to examine and diagnose the state of your application. With Memory Analyzer, you can inspect threads, objects, bundles, and whole data collections to debug Java code problems that go beyond memory leaks.

Snapshot dump types

Memory Analyzer can currently work with three dump types:

- **IBM Portable Heap Dump (PHD):** This proprietary IBM format contains only the type and size of each Java object in the process, and the relationships among the objects. This dump-file format is significantly smaller than the other formats and contains the least information. The data is usually sufficient, though, for diagnosing memory leaks and getting a basic understanding of the application's architecture and footprint.
- **HPROF binary dump:** The HPROF binary format contains all the data present in the IBM PHD format as well as the primitive data held inside the Java objects, and the thread details. You can look at the values held in fields inside the objects and see which methods were being executed at the time the dump was taken. The additional primitive data makes HPROF dumps significantly larger than PHD-format dumps; they are approximately the same size as the used Java heap.
- **IBM system dumps:** When the IBM Java runtime is being used, the native operating-system dump file — a core file on AIX® or Linux, a minidump on Windows®, or a SVC dump on z/OS® — can be loaded into Memory Analyzer. These dumps contain the entire memory image of the running application — all the information and data in the HPROF format, as well as all of the native-memory and thread information. This is the largest and most comprehensive dump-file format.

Both IBM dump types are available only with the Diagnostic Tool Framework for Java (DTFJ) plug-in installed (see Resources and the Memory Analyzer variants sidebar).

Table 1 summarises the differences among the dump-file types:

Table 1. Summary of the dump types' characteristics

Dump format	**Approxir size on disk**	**Objects, classes, and classloa**	**Thread details**	**Field names**	**Field and array referenc**	**Primitive fields**	**Primitive array contents**	**Accurate garbage- roots**	**Native memory and threads**
IBM PHD	20 percent of Java heap size	Y	With Javacore*	N	Y	N	N	N	N
HPROF	Java heap size	Y	Y	Y	Y	Y	Y	Y	N
IBM system dumps	Java heap size + 30 percent	Y	Y	Y	Y	Y	Y	Y	Y

*By loading in both a javacore.txt file (IBM thread dump file) and a heapdump.phd file that were generated at the same time, Memory Analyzer makes thread details available in the IBM PHD format dump.

Both the HPROF and IBM system dump formats can be compressed well, usually to around 20 percent of their original size, using operating-system tools.

Obtaining snapshot dumps

Different mechanisms are available for obtaining the various dumps for each of the Java runtimes, providing flexibility that lets you generate snapshot dumps for scenarios beyond those involving `OutOfMemoryErrors`. The mechanisms available depend on which vendor's Java runtime you're using.

Prerequisites

For all dump types, you must ensure sufficient disk space for the dumps so that they are not truncated. The default location of the dumps is the current working directory of the JVM process. For IBM JVMs, you can change this with the `-Xdump` file command-line option. For the HotSpot JVM, you can change it using the `-XX:HeapDumpPath` command-line option. See Resources for links to the relevant syntax.

Dumps from the operating system can be used for both IBM and HotSpot JVMs. For the IBM JVM, you can create dumps with the `jextract` tool (shipped with the JDK) and load them directly into Memory Analyzer; for the HotSpot JVM, you use the `jmap` tool to extract heap dumps from core dumps. (We discuss both techniques in detail later in this article.) However, on some operating systems, you must ensure that the process is running with sufficient `ulimits` before creating the core dump; otherwise, the core dump will be truncated and analysis will be limited. If the `ulimits` are incorrect, you must modify them and restart the process before gathering a dump. See Resources for links to detailed information on obtaining system dumps from AIX, Linux®, z/OS, and Solaris.

Obtaining a snapshot dump: HotSpot runtimes

The HotSpot-based Java runtimes generate the HPROF format dump only. You can choose among several interactive methods and one event-based method for generating the dump:

- Interactive methods:
  - **Using a Ctrl+Break**: If the `-XX:+HeapDumpOnCtrlBreak` command-line option is set for the running application, an HPROF format dump is generated along with a thread dump when a Ctrl+Break event, or `SIGQUIT` (usually generated using `kill -3`), is sent via the console. This option may not be available on some versions, in which case try:

```
-Xrunhprof:format=b,file=heapdump.hprof
```

  - **Using the `jmap` tool**: The `jmap` utility tool (see Resources), delivered in the bin directory of the JDK, provides an option to request an HPROF dump from the running process. With Java 5, use:

```
jmap -dump:format=b pid
```

    With Java 6, use this version, where `live` is optional and results in only the "live" objects being written to the dump-file process ID (PID):

```
jmap -dump[live,]format=b,file=filename pid
```

  - **Using the operating system**: Use the nondestructive `gcore` command or the destructive `kill -6` or `kill -11` commands to produce a core file. Then, extract a heap dump from the core file using `jmap`:

```
jmap -dump:format=b,file=heap.hprof path to java executable core
```

- **Using the JConsole tool:** A `dumpHeap` operation is provided under the `HotSpotDiagnostic` MBean in JConsole. This operation requests that a HPROF dump be generated.

- Event-based method:
    - **On an `OutOfMemoryError`:** If the `-XX:+HeapDumpOnOutOfMemoryError` command-line option is set for the running application, an HPROF format dump is generated when an `OutOfMemoryError` occurs. It is ideal to have this in place for production systems, because it is almost always required to diagnose memory issues, and it incurs no ongoing performance overhead. In older releases of HotSpot-based Java runtimes, there's no limit to how many heap dumps are produced on this event per JVM run; in newer releases, a maximum of one heap dump is produced on this event per JVM run.

Obtaining a snapshot dump: IBM runtimes

The IBM runtimes provide dump and trace engines that can generate either PHD-format or system dumps in a large number of interactive and event-based based scenarios. You can also generate interactive dumps using the Health Center tool or programmatically using a Java API.

- Interactive methods
    - **Using a `SIGQUIT` or Ctrl+Break:** When a Ctrl+Break or `SIGQUIT` (usually generated using `kill -3`) is sent to the IBM runtime, a user event is generated in the IBM dump engine. By default this event only generates a thread dump file (javacore.txt). You can use the `-Xdump:heap:events=user` option to generate a PHD-format dump, or the `-Xdump:system:events=user` option to generate a system dump of the Java application.

    - **Using the operating system to produce a system dump:**
        - AIX: `gencore` (or the destructive `kill -6` or `kill -11`)
        - Linux/Solaris: `gcore` (or the destructive `kill -6` or `kill -11`)
        - Windows: `userdump.exe`

        - z/OS: `SVCDUMP` or console dump
    - **Using IBM Monitoring and Diagnostics Tools for Java - Health Center:** The Health Center tool provides a menu option for requesting either a PHD or a system dump from a running Java process (see Resources).

- Event-based methods. The IBM dump and trace engines provide a flexible set of capabilities for generating PHD and system dumps on a large number of events, from exceptions being thrown to methods being executed. Using them, you should be able to generate dumps for most problem scenarios you want to diagnose:
    - **Using the IBM dump engine:** The dump engine provides a large number of events on which you can produce a PHD or system dump. Further, it lets you filter on types of those events in order to exercise finer-grained control over when to generate dumps.

      You can see the default events by using the `-Xdump:what` option. You'll notice, for example, that a heapdump.phd and javacore.txt are produced on the first four `OutOfMemoryError` exceptions in the JVM.

      To gather more data, you can produce a system dump instead of a heap dump on an `OutOfMemoryError` exception:

```
-Xdump:heap:none -Xdump:java+system:events=systhrow,
 filter=java/lang/OutOfMemoryError,range=1..4,request=exclusive+compact+prepwalk
```

Some exceptions, for example `NullPointerExceptions`, are generated commonly in most applications by a wide range of code. This makes it difficult to generate a dump on a particular `NullPointerException` of interest. To help you be more specific about which exception to generate the dump on, an extra level of filtering is provided for "throw" and "catch" events that lets you specify the throwing and catching methods, respectively. You do this by adding a # separator and then adding the throwing or catching method as appropriate. For example, this option produces a system dump when a `NullPointerException` is thrown by the `bad()` method:

```
-Xdump:system:events=throw,
       filter=java/lang/NullPointerException#com/ibm/example/Example.bad
```

This option produces a system dump when a `NullPointerException` is caught by the `catch()` method:

```
-Xdump:system:events=catch,
       filter=java/lang/NullPointerException#com/ibm/example/Example.catch
```

In addition to filtering on the events, you can also specify a range of events on which you want dumps to be generated. For example, this option produces a dump only on the fifth occurrence of a `NullPointerException`:

```
-Xdump:system:events=throw, filter=java/lang/NullPointerException,range=5
```

This option uses a range to produce a dump only on the second, third, and fourth occurrences of a `NullPointerException`:

```
-Xdump:system:events=throw, filter=java/lang/NullPointerException,range=2..4
```

Table 2 summarizes the most useful events and filters:

Table 2. Available dump events

Event	Descri	Availal filterin	Example
`gpf`	General protection fault (crash)		`-Xdump:system:ever`
`user`	User generated signal (`SIGQUIT` or Ctrl+Break)		`-Xdump:system:ever`
`vmstop`	VM shutdown, including call to `System.exit()`	exit code	`-Xdump:system:ever` Generate a system dump on VM shutdown with an exit code between `0` and `10`.
`load`	Class load	Class name	`-Xdump:system:ever` Generate a system

			dump when the `com.ibm.example.Ex` class is loaded.
`unload`	Class unload	Class name	`-Xdump:system:ever` Generate a system dump when the `com.ibm.example.Ex` class is unloaded.
`throw`	An exception being thrown	Exception class name	`-Xdump:system:ever` Generate a system dump when a `ConnectException` is generated.
`catch`	An exception being caught	Exception class name	`-Xdump:system:ever` Generate a system dump when a `ConnectException` is caught.
`systhrow`	A Java exception is about to be thrown by the JVM. (This is different from the	Exception class name	`-Xdump:system:ever` Generate a system dump when an `OutOfMemoryError` is generated.

	`throw` event because it is only triggered for error conditions detected internally in the JVM.)		
`allocation`	A Java object is allocated	Size of object being allocated	`-Xdump:system:ever` Generate a system dump when an object larger than 5MB is allocated.

- **Using the IBM trace engine:** The trace engine allows PHD and system dumps to be triggered on method entry or exit for any Java method running in the application. You accomplish this by using the `trigger` keyword to the `-Xtrace` command-line options that control the IBM trace engine. The syntax for the trigger option is:

```
method{methods[,entryAction[,exitAction[,delayCount[,matchcount]]]]}
```

Adding the following command-line option to the application produces a system dump when the `Example.trigger()` method is called:

```
-Xtrace:maximal=mt,trigger=method{com/ibm/example/Example.trigger,sysdump}
```

This command-line option produces a PHD dump when the `Example.trigger()` method is called:

```
-Xtrace:maximal=mt,trigger=method{com/ibm/example/Example.trigger,heapdump}
```

It is recommended, though, that you set a range so that you don't create dumps every time the method is called. This example ignores the first five calls to `Example.trigger()` and then triggers one dump:

```
-Xtrace:maximal=mt,trigger=method{com/ibm/example/Example.trigger,sysdump,,5,1}
```

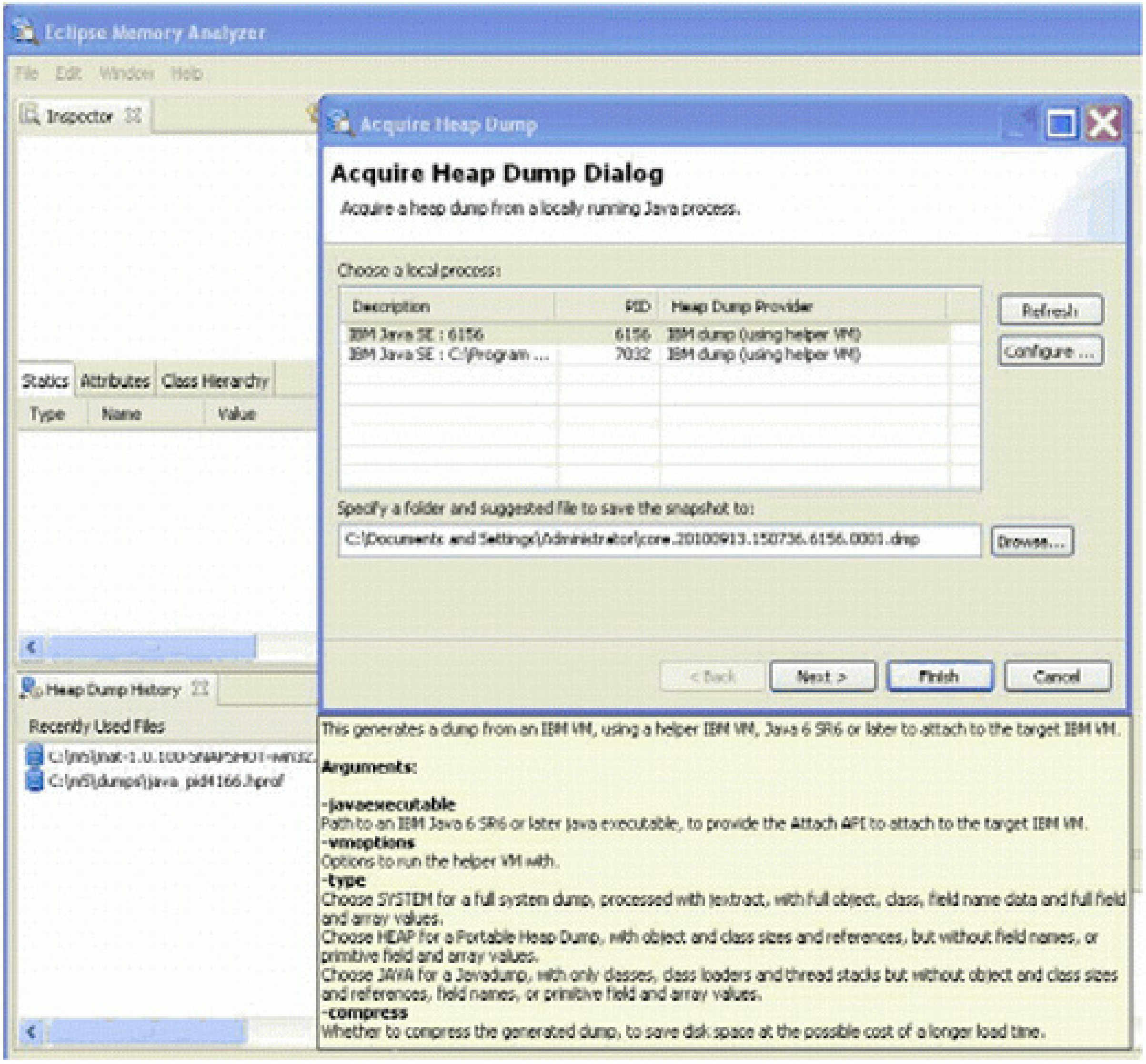
Note that an empty term is used for the `exitAction` in this example because we're triggering the dumps on method entry only.

- **Programmatic methods:** The IBM runtimes also provide a `com.ibm.jvm.Dump` class with `javaDump()`, `heapDump()`, and `systemDump()` methods. They generate thread dumps, PHD dumps, and system dumps, respectively.

Acquiring a dump using Memory Analyzer

As well as the methods for obtaining dumps that are provided by the runtimes themselves, Memory Analyzer also provides an Acquire Heap Dump option, shown in Figure 1, that allows you to trigger and load a snapshot dump from a Java process running on the same machine as Memory Analyzer:

Figure 1. Using the Acquire Heap Dump function in Memory Analyzer



Acquire Heap Dump Dialog

Acquire a heap dump from a locally running Java process.

Choose a local process:

Description	PID	Heap Dump Provider
IBM Java SE : 6156	6156	IBM dump (using helper VM)
IBM Java SE : C:\Program ...	7032	IBM dump (using helper VM)

Specify a folder and suggested file to save the snapshot to:

C:\Documents and Settings\Administrator\core.20100913.150736.6156.0001.dmp

This generates a dump from an IBM VM, using a helper IBM VM, Java 6 SR6 or later to attach to the target IBM VM.

Arguments:

-javaexecutable
Path to an IBM Java 6 SR6 or later java executable, to provide the Attach API to attach to the target IBM VM.
-vmoptions
Options to run the helper VM with.
-type
Choose SYSTEM for a full system dump, processed with jextract, with full object, class, field name data and full field and array values.
Choose HEAP for a Portable Heap Dump, with object and class sizes and references, but without field names, or primitive field and array values.
Choose JAVA for a Javadump, with only classes, class loaders and thread stacks but without object and class sizes and references, field names, or primitive field and array values.
-compress
Whether to compress the generated dump, to save disk space at the possible cost of a longer load time.

On HotSpot-based runtimes, Memory Analyzer generates the dump using `jmap`. For the IBM runtimes, the dump is generated using the Java "late attach" functionality and programmatic API. Java 6 SR6 is required for the function to work, because earlier releases do not contain the "late attach" function.

Postprocessing requirements

For IBM system dumps, the dump must be postprocessed using the `jextract` tool shipped with the JDK:

```
jextract core
```

Ideally, `jextract` is run on the same physical machine that produced the dump, using `jextract` from the same JDK installation that produced the dump, and with read access to the same libraries that `java` process was running with. Given that `jextract` can consume significant CPU cycles processing the dump, this may be unacceptable in some production systems. In this case, the dump should be processed on the closest matching system, such as a preproduction test system. The Service Refresh (SR) and Fix Pack (FP) versions of the Java runtimes should match.

`jextract` produces a ZIP file that includes the original core dump, a processed representation of the dump, the Java executable, and the libraries used by the `java` process. You can delete the original (unzipped) core dump after running `jextract`. The ZIP file is what you should load into Memory Analyzer.

You can extract a PHD dump from a `jextracted` system dump by loading the ZIP into `jdmpview` and executing the `heapdump` command (see Resources).

Using Memory Analyzer to analyse problems

Memory Analyzer can diagnose `OutOfMemoryErrors` by looking for areas of the application that are either leaking memory or have a footprint requirement that's too large for the available memory. Memory Analyzer does automatic leak detection and generates a Leak Suspects report (see Resources).

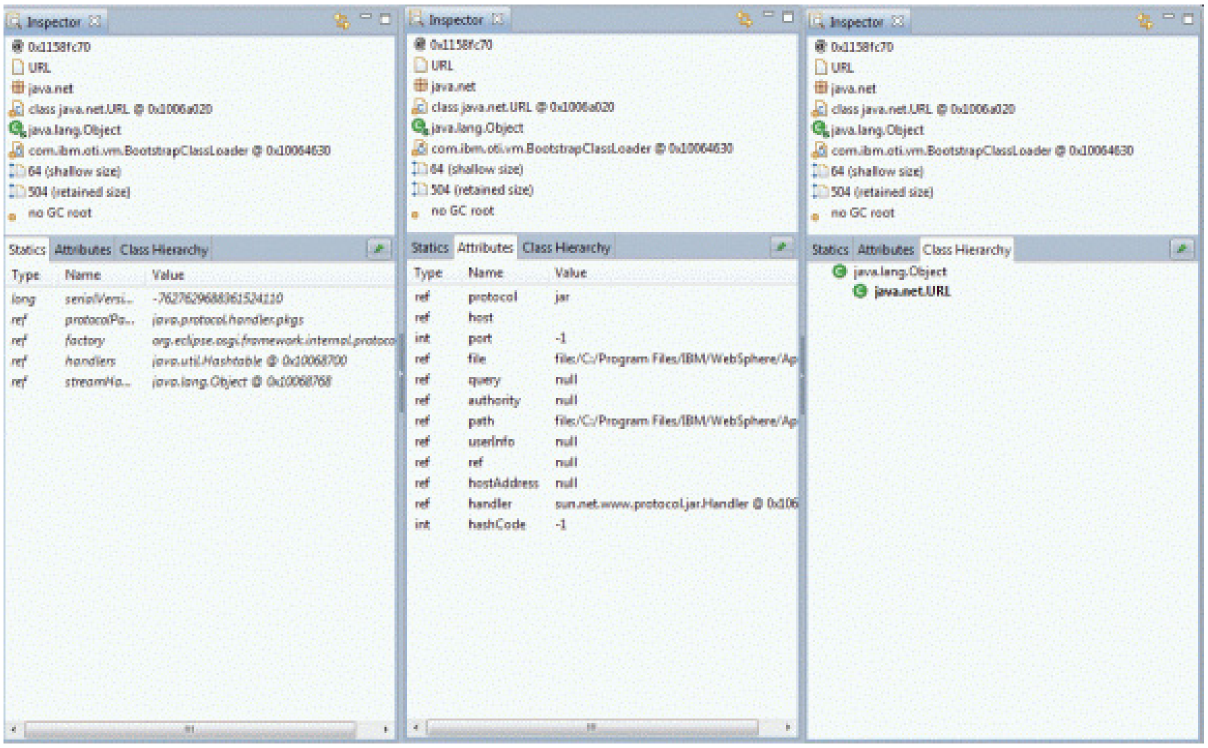
The additional data that's available in the HPROF and IBM system dumps, particularly the field names and field values — along with the capabilities of the Inspector view and Object Query Language (OQL) — also make it possible to diagnose a wider range of problem types than "What's using all of the memory?". For example, you can ascertain the occupancy and load factor of collections to see if they are efficiently sized, or look at the hostname and port associated with a `ConnectException` to see what connection the application was trying to create.

Looking at fields in an object with the Inspector

When any object is selected in Memory Analyzer, the Inspector view shows the available information relating to that object, including the class hierarchy, attributes, and statics. The Attributes panel shows the instance fields and values associated with the object, and the Statics panel shows the static fields and values associated with the class.

The Inspector view shown in Figure 2 for a simple `java.net.URL` object lets you see details about that object, including the type of protocol the URL is for and the destination:

Figure 2. The Statics, Attributes, and Class Hierarchy panels in the Inspector view



0x1158fc70
URL
java.net
class java.net.URL @ 0x1006a020
java.lang.Object
com.ibm.oti.vm.BootstrapClassLoader @ 0x10064630
64 (shallow size)
504 (retained size)
no GC root

Type	Name	Value
long	serialVersi...	-7627629688361524110
ref	protocolPa...	java.protocol.handler.pkgs
ref	factory	org.eclipse.osgi.framework.internal.protoco
ref	handlers	java.util.Hashtable @ 0x10068700
ref	streamHa...	java.lang.Object @ 0x10068768

0x1158fc70
URL
java.net
class java.net.URL @ 0x1006a020
java.lang.Object
com.ibm.oti.vm.BootstrapClassLoader @ 0x10064630
64 (shallow size)
504 (retained size)
no GC root

Type	Name	Value
ref	protocol	jar
ref	host	
int	port	-1
ref	file	file:/C:/Program Files/IBM/WebSphere/Ap
ref	query	null
ref	authority	null
ref	path	file:/C:/Program Files/IBM/WebSphere/Ap
ref	userInfo	null
ref	ref	null
ref	hostAddress	null
ref	handler	sun.net.www.protocol.jar.Handler @ 0x106
int	hashCode	-1

0x1158fc70
URL
java.net
class java.net.URL @ 0x1006a020
java.lang.Object
com.ibm.oti.vm.BootstrapClassLoader @ 0x10064630
64 (shallow size)
504 (retained size)
no GC root

java.lang.Object
java.net.URL

In Figure 2, you can see in the Attributes panel that the URL object refers to a JAR file (the protocol field) located on the local file system (in the location specified by the path and file fields).

Running queries against the objects using OQL

OQL can be used to query a dump using custom, SQL-like queries. This topic could be an article in itself, so we'll just highlight a few examples. For more details, consult the Help contents on OQL available from within Memory Analyzer.

OQL is particularly useful for following a path down the outgoing references and fields of a set of objects to a particular field. For example, if class A has a field `foo` of type B, and class B has a field `bar` that is a `String`, then a simple query to find all of those `Strings` would be:

```
SELECT aliasA.foo.bar.toString()
FROM A aliasA
```

We give class A an alias, `aliasA`, which we then reference in the `SELECT` clause. This query strictly selects only from instances of class A. If we wanted to select from all instances of class A as well as any subclasses, we'd use:

```
SELECT aliasA.foo.bar.toString()
FROM INSTANCEOF A aliasA
```

Here is a more complicated example with `DirectByteBuffers`:

```
SELECT k, k.capacity
FROM java.nio.DirectByteBuffer k
WHERE ((k.viewedBuffer=null)and(inbounds(k).length>1))
```

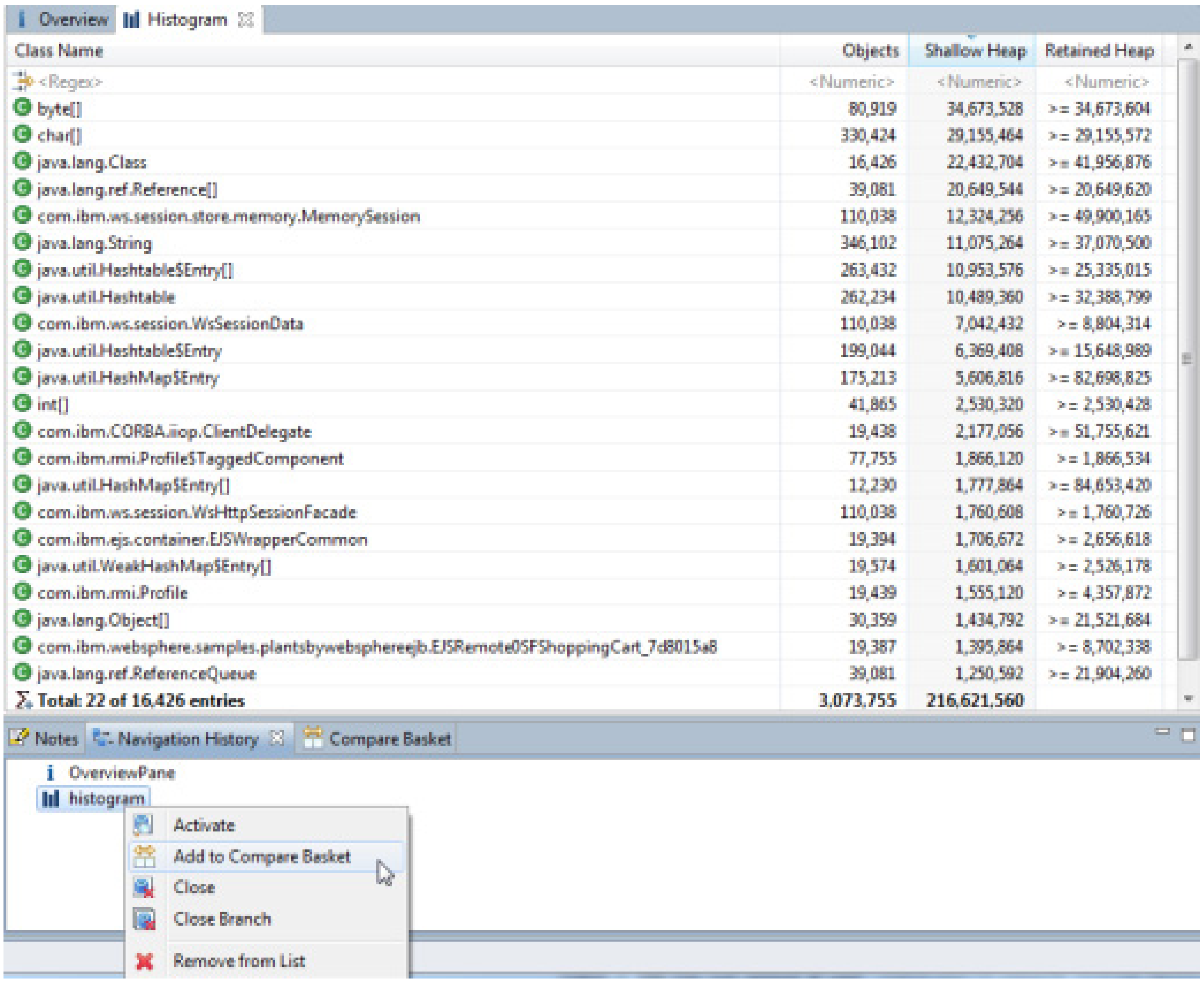
In this case, we want to get the capacity field of any `DirectByteBuffer`, which gives the native memory held by that object. We also want to filter out any `DirectByteBuffers` that have a null `viewedBuffer` field (because those are just views into other `DirectByteBuffers`) and more than one inbound reference (so that we don't look at those pending clean-up with their phantom reference — that is, we want only "live" `DirectByteBuffers`).

Running comparisons between views or dumps

With Memory Analyzer, you can compare tables generated by queries. The tables can either be from the same dump, letting you see whether `String` objects from one view are present in a collection object seen in another view, or across separate dumps, letting you look for changes in data, for example growth of object collections.

To run a comparison, you add the relevant tables to the Compare Basket and then request that the entries in the basket be compared. First find and select the entry for the table in the Navigation History, then select **Add to Compare Basket** from the context menu, as shown in Figure 3:

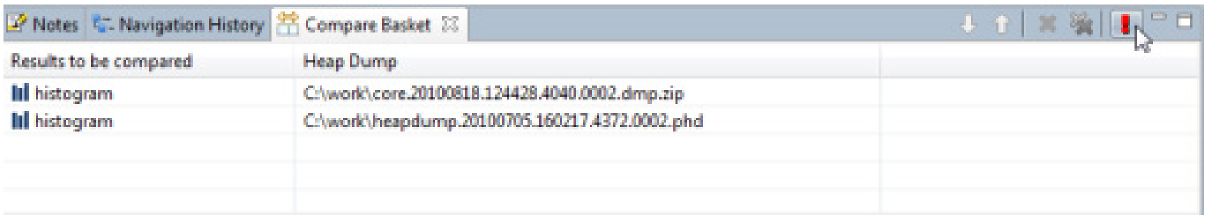
Figure 3. Adding tables from the Navigation History view to the Compare Basket



Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
byte[]	80,919	34,673,528	>= 34,673,604
char[]	330,424	29,155,464	>= 29,155,572
java.lang.Class	16,426	22,432,704	>= 41,956,876
java.lang.ref.Reference[]	39,081	20,649,544	>= 20,649,620
com.ibm.ws.session.store.memory.MemorySession	110,038	12,324,256	>= 49,900,165
java.lang.String	346,102	11,075,264	>= 37,070,500
java.util.Hashtable$Entry[]	263,432	10,953,576	>= 25,335,015
java.util.Hashtable	262,234	10,489,360	>= 32,388,799
com.ibm.ws.session.WsSessionData	110,038	7,042,432	>= 8,804,314
java.util.Hashtable$Entry	199,044	6,369,408	>= 15,648,989
java.util.HashMap$Entry	175,213	5,606,816	>= 82,698,825
int[]	41,865	2,530,320	>= 2,530,428
com.ibm.CORBA.iiop.ClientDelegate	19,438	2,177,056	>= 51,755,621
com.ibm.rmi.Profile$TaggedComponent	77,755	1,866,120	>= 1,866,534
java.util.HashMap$Entry[]	12,230	1,777,864	>= 84,653,420
com.ibm.ws.session.WsHttpSessionFacade	110,038	1,760,608	>= 1,760,726
com.ibm.ejs.container.EJSWrapperCommon	19,394	1,706,672	>= 2,656,618
java.util.WeakHashMap$Entry[]	19,574	1,601,064	>= 2,526,178
com.ibm.rmi.Profile	19,439	1,555,120	>= 4,357,872
java.lang.Object[]	30,359	1,434,792	>= 21,521,684
com.ibm.websphere.samples.plantsbywebsphereejb.EJSRemote0SFShoppingCart_7d8015a8	19,387	1,395,864	>= 8,702,338
java.lang.ref.ReferenceQueue	39,081	1,250,592	>= 21,904,260
Total: 22 of 16,426 entries	**3,073,755**	**216,621,560**	

Once you have two entries in the Compare Basket, you can run the comparison using the Compare-the-results button (the red exclamation mark) in the top right-hand corner of the panel, as shown in Figure 4:

Figure 4. Comparing the results of the entries in the Compare Basket



Results to be compared	Heap Dump
histogram	C:\work\core.20100818.124428.4040.0002.dmp.zip
histogram	C:\work\heapdump.20100705.160217.4372.0002.phd

Footprint and memory efficiency

Another important use of Memory Analyzer is to find which components are using most of the heap, even in situations without a memory leak. If memory usage can be reduced, then the system's capacity or performance can be improved, allowing more sessions or less time spent garbage collecting.

The Top Components report is the first step. It splits memory usage by component in the system, analyzes the usage in each component, and looks for wasteful practices. Objects dominated (retained by) another object can be said to be *owned* by that dominator. The Top Components report lists all the objects that are not owned by another object. These are the top dominators of the heap. The top dominators are then divided by classloader using the class of the objects, and all those top dominators and objects owned by them are allocated to appropriate classloaders. You analyse further by selecting one of the classloaders in the report to open a new classloader-specific component report.

For each component, the Collections objects are analyzed. The Collections classes, as shown by `java.util.*`, are a great time-saver for programmers, providing well-tested implementations of lists, sets, and maps. The average application can have millions of collections, so wasted space in collections can be significant.

Empty collections are one common cause of wasted memory. `ArrayLists`, `Vectors`, `HashMaps`, and `HashSets` are created with a default-size backing array of perhaps 10 entries, ready to hold entries. It's surprisingly common in applications for a collection to be created but no objects stored in it. This can rapidly consume memory. For example, with 100,000 empty collections, the backing arrays alone could consume 100,000 * (24+10*4) bytes = 6MB.

The Empty Collection report looks at the standard collection classes and extensions thereof and analyzes them by size of collection. It then produces a table for each collection sorted by size of collection, with the most frequent size first. If a large proportion of instances of a type of a collection are empty, then the report flags this as a possible memory waste.

One solution is to delay allocating the collection until an entry is needed to be inserted. Another is to allocate a collection with a default size of 0 or 1, letting it grow if required at some runtime cost. A third is to trim the collection to size after the initialization phase is complete.

A related area is collections with only a few entries and a lot of wasted space. The Collection Fill Ratio section shows, for each collection type, the number of instances of that collection with a particular fill ratio. This reveals collections with a large proportion of empty space.

Duplicate strings

Strings and character arrays occupy a large amount of space in a typical business application, so they are another area worthy of analysis. This section of the component report analyses strings for common content. Strings are immutable. Strings constants with the same value are guaranteed by the VM specification to use the same instance. Dynamically built strings have no such guarantee, and two `Strings` built by, for example, reading data from a database or disk that have the same value will have separate instances and separate backing character arrays. If these strings are kept, then this can be significant.

You can solve this problem by using `String.intern()` or maintaining a user hash set or hash map.

Wasted char arrays

`String.substring()` is implemented in the Java language by building a new `String` sharing the original character array. This is efficient if the original string is still needed. If only a small substring is needed, then — because the whole character array is retained — some space is wasted. The Waste In Char Arrays query shows the amount of wasted space in character arrays that are only referenced by strings.

Eclipse bundles and classloader hierarchy

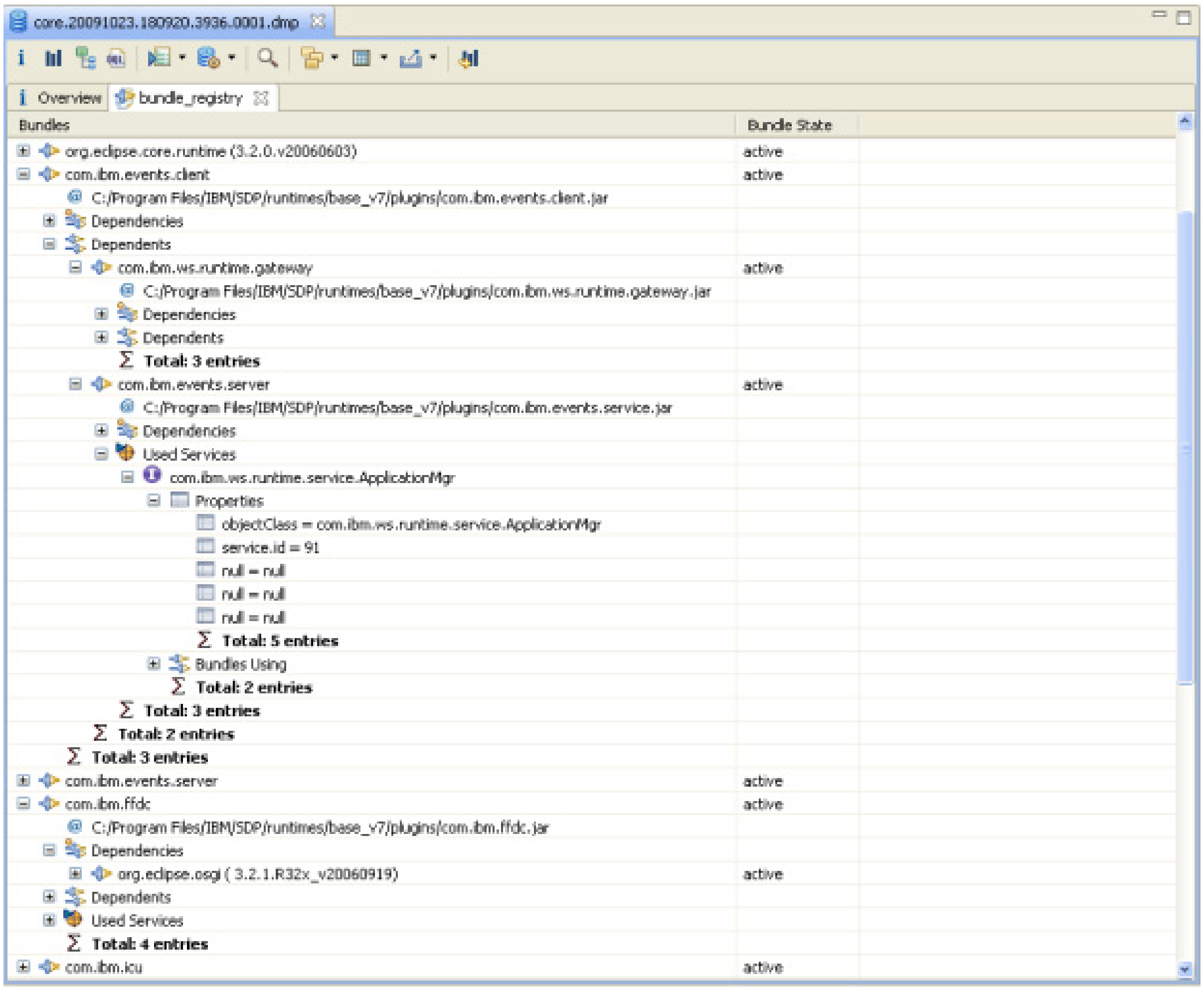
Modern applications are divided into components, often based on classloaders, to provide some degree of isolation between parts of the application. Components can be updated by stopping the usage of one component classloader and loading a new version of the component using a new classloader. In time, the old version is freed by garbage collection, assuming the application contains no external references to classes or objects or the classloader.

The Class Loader Explorer query shows all the classloaders in the system, and so works for all applications. It shows the classes loaded by a classloader and also the parent chain of the classloader so that classloading problems can be understood. By inspection, you can see if multiple copies of a classloader exist. If next to no instances of classes are defined by a classloader, then it is likely that the classloader is idle.

The duplicate-classes query shows class names that have been loaded by multiple classloaders. This could be an indication of a classloader memory leak. It takes just one reference to an object held elsewhere in the system, such as in a registry, for a classloader leak to occur. The object holds a reference to its class, and the class to the classloader, and the classloader to all the defined classes.

A common classloading framework is the OSGi framework. One implementation is Eclipse Equinox, used for Eclipse-based applications to separate the plug-ins, and also used for WebSphere® Application Server 6.1 and later versions. When trying to understand the state of an application, it is useful to know the state of all the bundles. The Eclipse Equinox Bundle Explorer query, shown in Figure 5, does just that:

Figure 5. Eclipse Bundle Explorer



```
core.20091023.180920.3936.0001.dmp
Overview | bundle_registry

Bundles                                                                          Bundle State
⊞ org.eclipse.core.runtime (3.2.0.v20060603)                                     active
⊟ com.ibm.events.client                                                          active
    C:/Program Files/IBM/SDP/runtimes/base_v7/plugins/com.ibm.events.client.jar
  ⊞ Dependencies
  ⊟ Dependents
    ⊟ com.ibm.ws.runtime.gateway                                                 active
        C:/Program Files/IBM/SDP/runtimes/base_v7/plugins/com.ibm.ws.runtime.gateway.jar
      ⊞ Dependencies
      ⊞ Dependents
      Σ Total: 3 entries
    ⊟ com.ibm.events.server                                                      active
        C:/Program Files/IBM/SDP/runtimes/base_v7/plugins/com.ibm.events.service.jar
      ⊞ Dependencies
      ⊟ Used Services
        ⊟ com.ibm.ws.runtime.service.ApplicationMgr
          ⊟ Properties
              objectClass = com.ibm.ws.runtime.service.ApplicationMgr
              service.id = 91
              null = null
              null = null
              null = null
              Σ Total: 5 entries
          ⊞ Bundles Using
          Σ Total: 2 entries
      Σ Total: 3 entries
    Σ Total: 2 entries
  Σ Total: 3 entries
⊞ com.ibm.events.server                                                          active
⊟ com.ibm.ffdc                                                                   active
    C:/Program Files/IBM/SDP/runtimes/base_v7/plugins/com.ibm.ffdc.jar
  ⊟ Dependencies
    ⊞ org.eclipse.osgi ( 3.2.1.R32x_v20060919)                                   active
  ⊞ Dependents
  ⊞ Used Services
  Σ Total: 4 entries
⊞ com.ibm.icu                                                                    active
```

A system or HPROF dump has all the objects and fields. The Bundle Explorer shows all the bundles in the system, together with their states and dependencies, dependents, and services. It can show bundles that are unexpectedly active and so using more resources.

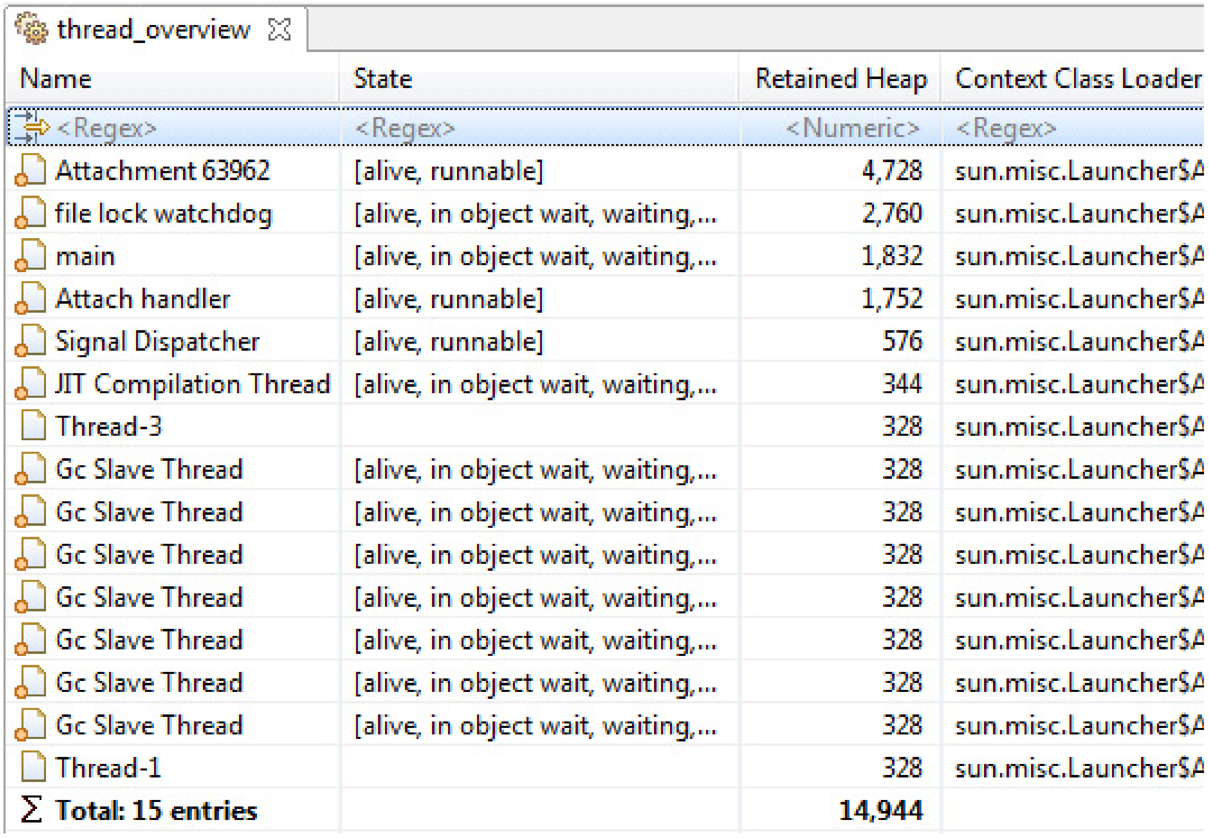
Thread data usage

As indicated in Table 1, a dump can include thread details that can provide unique insights into what was happening at the time of the dump. This can include all active threads stacks, all of the frames for each thread, and most important, some or all of the active Java locals on those frames.

Thread Overview view

The Thread Overview view, shown in Figure 6, shows every thread in the JVM as well as various attributes of that thread, such as its retained heap size, context classloader, priority, state, and native ID:

Figure 6. Thread Overview



thread_overview

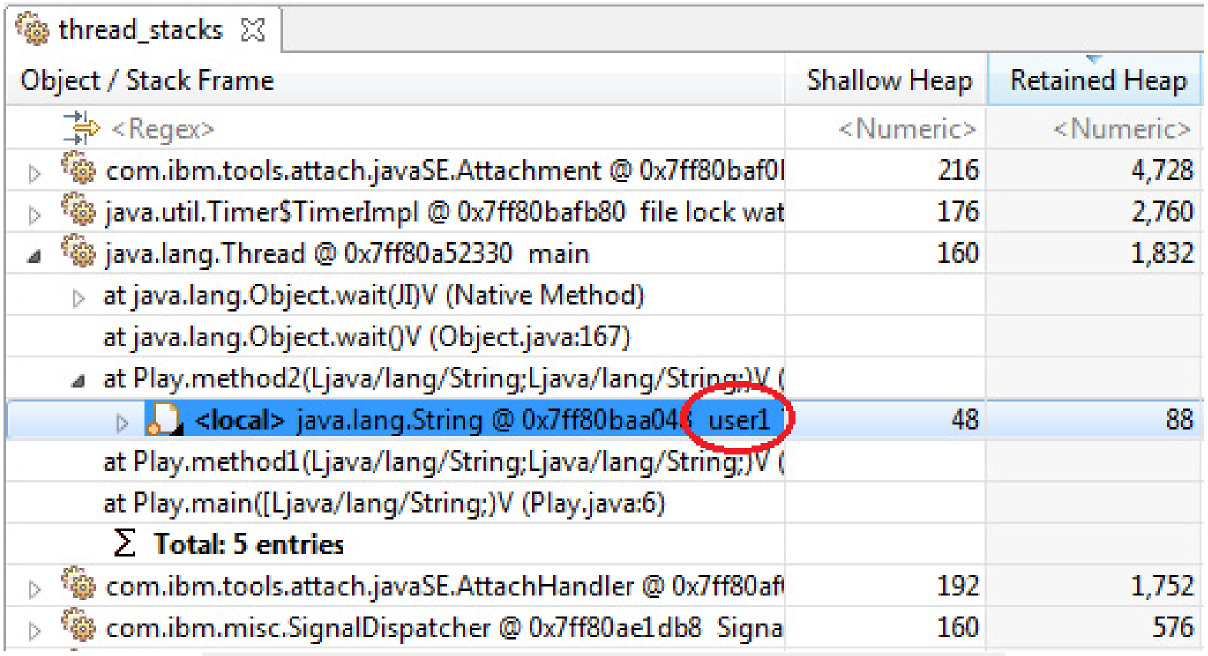
Name	State	Retained Heap	Context Class Loader
<Regex>	<Regex>	<Numeric>	<Regex>
Attachment 63962	[alive, runnable]	4,728	sun.misc.Launcher$A
file lock watchdog	[alive, in object wait, waiting,...	2,760	sun.misc.Launcher$A
main	[alive, in object wait, waiting,...	1,832	sun.misc.Launcher$A
Attach handler	[alive, runnable]	1,752	sun.misc.Launcher$A
Signal Dispatcher	[alive, runnable]	576	sun.misc.Launcher$A
JIT Compilation Thread	[alive, in object wait, waiting,...	344	sun.misc.Launcher$A
Thread-3		328	sun.misc.Launcher$A
Gc Slave Thread	[alive, in object wait, waiting,...	328	sun.misc.Launcher$A
Gc Slave Thread	[alive, in object wait, waiting,...	328	sun.misc.Launcher$A
Gc Slave Thread	[alive, in object wait, waiting,...	328	sun.misc.Launcher$A
Gc Slave Thread	[alive, in object wait, waiting,...	328	sun.misc.Launcher$A
Gc Slave Thread	[alive, in object wait, waiting,...	328	sun.misc.Launcher$A
Gc Slave Thread	[alive, in object wait, waiting,...	328	sun.misc.Launcher$A
Gc Slave Thread	[alive, in object wait, waiting,...	328	sun.misc.Launcher$A
Thread-1		328	sun.misc.Launcher$A
Σ **Total: 15 entries**		**14,944**	

The retained heap size is particularly useful in cases in which no Java heap problem per se occurs during an `OutOfMemoryError`, but rather the sum of the retained heaps of the threads is "too much." In this case, the JVM may be undersized, the thread pool sizes may be too large, or the average or maximum Java heap "load" of a thread is too high.

Thread Stacks view

The Thread Stacks view, shown in Figure 7, shows every thread, its stack, stack frames, and Java locals on those stack frames:

Figure 7. Thread Stacks view



thread_stacks

Object / Stack Frame	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
com.ibm.tools.attach.javaSE.Attachment @ 0x7ff80baf0l	216	4,728
java.util.Timer$TimerImpl @ 0x7ff80bafb80 file lock wat	176	2,760
java.lang.Thread @ 0x7ff80a52330 main	160	1,832
at java.lang.Object.wait(JI)V (Native Method)		
at java.lang.Object.wait()V (Object.java:167)		
at Play.method2(Ljava/lang/String;Ljava/lang/String;)V (		
<local> java.lang.String @ 0x7ff80baa04 user1	48	88
at Play.method1(Ljava/lang/String;Ljava/lang/String;)V (		
at Play.main([Ljava/lang/String;)V (Play.java:6)		
Σ Total: 5 entries		
com.ibm.tools.attach.javaSE.AttachHandler @ 0x7ff80af	192	1,752
com.ibm.misc.SignalDispatcher @ 0x7ff80ae1db8 Signa	160	576

Thread Details view

In both the Thread Overview and Thread Stacks views, you can right-click on a thread and select **Thread Details** either at the top of the menu or through Java Basics > Thread Details. This view gives more detailed information such as, if available, the native stack.

In the example in Figure 7, a thread of the type `java.lang.Thread` and with the name `main` — the main thread in a simple command-line program — is expanded. Each stack frame of the thread is displayed, and those with available Java locals are expandable. In this case, a `String` has been passed as an argument from `Play.method1` to `Play.method2`, and the contents of the string, `user1`, is highlighted in the red circle. You can imagine the power of being able to reconstruct or reverse-engineer what was happening at the time of the dump based on what was happening in each thread stack frame and on which objects.

Note that because of runtime optimization, not all related objects such as method parameters or object instances will be available (although those objects will be in the dump), but those objects that are actively being "worked on" usually will be.

Exception analysis

When exceptions are generated in the application, additional complications can make it more difficult to analyse the exception's cause. Two examples of this dilemma are:

- The use of logging mechanisms means that the exception is lost or the exception message is removed.

- The exception is producing a message that contains insufficient information.

In the first case, either the exception message or the entire exception is completely lost, making it difficult to know that the problem exists or to gain basic information about it. In the second case, the exception has been logged and the exception message and stack trace is available, but it doesn't contain the necessary information to resolve the exception's cause.

Because Memory Analyzer has access to the fields inside the objects, it's possible to find the exception message from the exception object. In some cases, it's also possible to extract additional data that's not in the original exception.

Locating exceptions in the snapshot dump

One way to locate the exceptions present in the snapshot dump is to use the OQL capability in Memory Analyzer to locate objects of interest in the dump. For example, this query finds all of the exception objects:

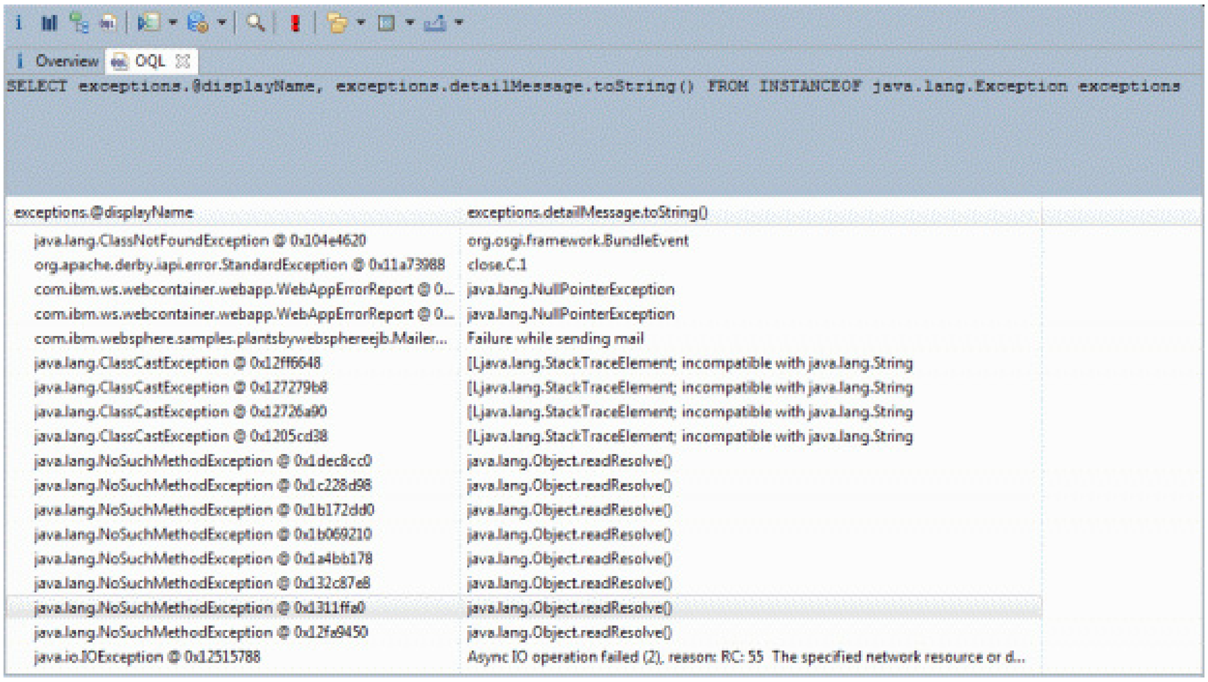
```
SELECT *
FROM INSTANCEOF java.lang.Exception exceptions
```

This next query produces a list of all of the exceptions, from which you can use the Inspector view to look at the fields inside each exception. Knowing that the field that contains the exception message is the `detailMessage` field, you can also modify the query to extract the exception messages directly and display them immediately as part of the results table:

```
SELECT exceptions.@displayName, exceptions.detailMessage.toString()
FROM INSTANCEOF java.lang.Exception exceptions
```

The preceding query produces the output shown in Figure 8:

Figure 8. Output of the OQL query for exceptions, including exception messages



```
SELECT exceptions.@displayName, exceptions.detailMessage.toString() FROM INSTANCEOF java.lang.Exception exceptions
```

exceptions.@displayName	exceptions.detailMessage.toString()
java.lang.ClassNotFoundException @ 0x104e4620	org.osgi.framework.BundleEvent
org.apache.derby.iapi.error.StandardException @ 0x11a73988	close.C.1
com.ibm.ws.webcontainer.webapp.WebAppErrorReport @ 0...	java.lang.NullPointerException
com.ibm.ws.webcontainer.webapp.WebAppErrorReport @ 0...	java.lang.NullPointerException
com.ibm.websphere.samples.plantsbywebsphereejb.Mailer...	Failure while sending mail
java.lang.ClassCastException @ 0x12ff6648	[Ljava.lang.StackTraceElement; incompatible with java.lang.String
java.lang.ClassCastException @ 0x127279b8	[Ljava.lang.StackTraceElement; incompatible with java.lang.String
java.lang.ClassCastException @ 0x12726a90	[Ljava.lang.StackTraceElement; incompatible with java.lang.String
java.lang.ClassCastException @ 0x1205cd38	[Ljava.lang.StackTraceElement; incompatible with java.lang.String
java.lang.NoSuchMethodException @ 0x1dec8cc0	java.lang.Object.readResolve()
java.lang.NoSuchMethodException @ 0x1c228d98	java.lang.Object.readResolve()
java.lang.NoSuchMethodException @ 0x1b172dd0	java.lang.Object.readResolve()
java.lang.NoSuchMethodException @ 0x1b069210	java.lang.Object.readResolve()
java.lang.NoSuchMethodException @ 0x1a4bb178	java.lang.Object.readResolve()
java.lang.NoSuchMethodException @ 0x132c87e8	java.lang.Object.readResolve()
java.lang.NoSuchMethodException @ 0x1311ffa0	java.lang.Object.readResolve()
java.lang.NoSuchMethodException @ 0x12fa9450	java.lang.Object.readResolve()
java.io.IOException @ 0x12515788	Async IO operation failed (2), reason: RC: 55 The specified network resource or d...

Figure 8 shows every exception that is still present in the application, and the message that would have been shown when the exception was thrown.

Extracting additional information relating to exceptions

Although finding the exception object from the dump allows you to recover the exception message, sometimes the exception message is too generic or vague to enable you to understand the problem's cause. A good example is the `java.net.ConnectException`. When trying to make a socket connection to a host that is not accepting connections, you get the following message:

```
java.net.ConnectException: Connection refused: connect
     at java.net.PlainSocketImpl.socketConnect(Native Method)
     at java.net.PlainSocketImpl.doConnect(PlainSocketImpl.java:352)
     at java.net.PlainSocketImpl.connectToAddress(PlainSocketImpl.java:214)
     at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:201)
     at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:377)

     at java.net.Socket.connect(Socket.java:530)
     at java.net.Socket.connect(Socket.java:480)
     at java.net.Socket.(Socket.java:377)
     at java.net.Socket.(Socket.java:220)
```

This message is sufficient if you have access to the code that's creating the socket and can see from the code which hostname and port are being used. In the case of more-complex code in which the hostname and port values are subject to change because they are obtained from external sources (user input values, databases, and the like), the message doesn't help you to understand why the connection is being refused.

The stack trace should include a socket object that contains the useful data, and if we can find the socket object in the snapshot dump using Memory Analyzer, then we can find out which hostname and port refused the connection.

The easiest way to do this is to generate a dump when the exception is thrown. This can be done on the IBM runtimes using the following `-Xdump` option set:

```
-Xdump:system:events=throw,range=1..1,
       filter=java/net/ConnectException#java/net/PlainSocketImpl.socketConnect
```

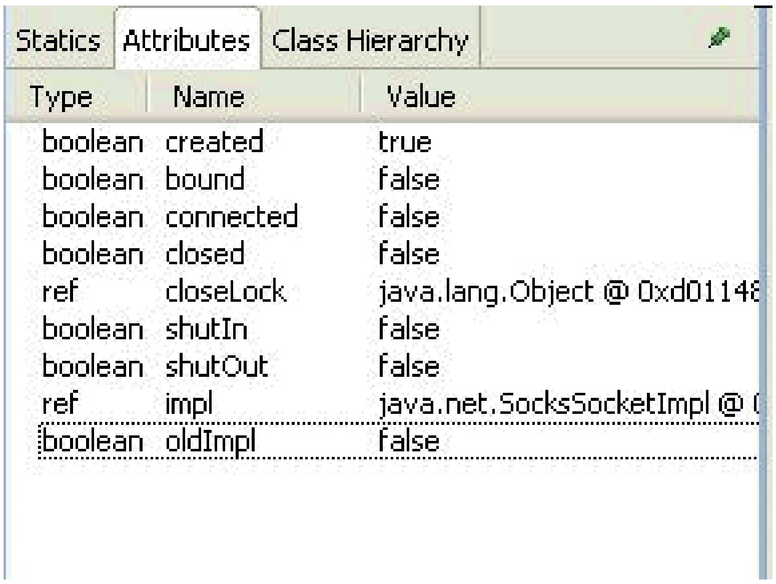
This option generates an IBM system dump on the first occurrence of a `ConnectException` generated by the `PlainSocketImpl.socketConnect()` method.

After loading the generated snapshot dump into Memory Analyzer, we can use the Open Query Browser > Java Basics > Thread Stacks option to list the threads and the objects associated with each method in the stack trace of the threads.

By expanding the current thread and the method frames in the thread, you can look at the objects associated with those methods. In the case of a `java.net.ConnectException`, the most interesting method is `java.net.Socket.connect()`. Expanding this method frame shows a reference to a `java.net.Socket` object in memory. This is the socket connection we were trying to make.

When the `Socket` object is selected, the fields are shown in the Inspector view, as you can see in Figure 9:

Figure 9. Inspector view for the Socket object

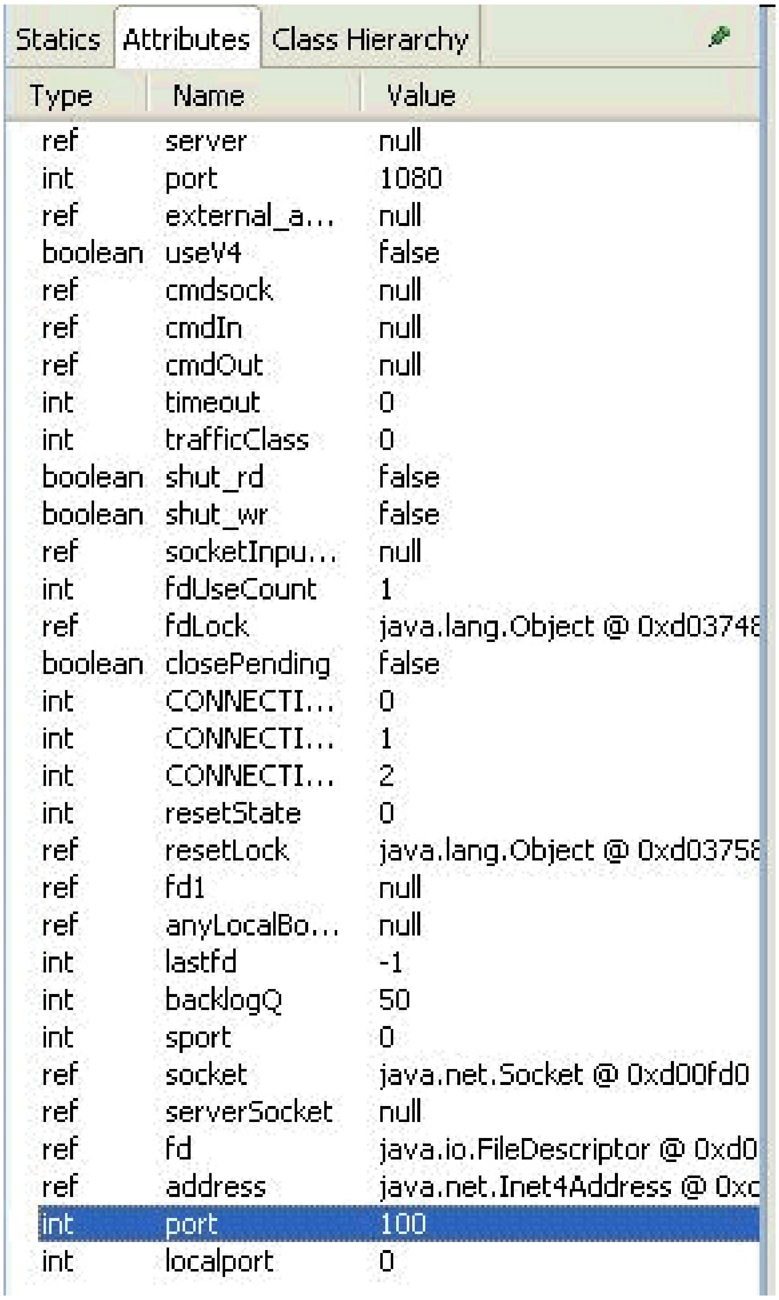


Type	Name	Value
boolean	created	true
boolean	bound	false
boolean	connected	false
boolean	closed	false
ref	closeLock	java.lang.Object @ 0xd0114
boolean	shutIn	false
boolean	shutOut	false
ref	impl	java.net.SocksSocketImpl @
boolean	oldImpl	false

The information in Figure 9 isn't too useful, because the real implementation of the

Socket is in the impl field. You can inspect the contents of the impl object by either expanding the Socket object and selecting the impl java.net.SocksSocketImpl line in the main panel, or by right-clicking on the impl field in the Inspector view and selecting **Go Into**. Now the fields for SocksSocketImpl are visible in the Inspector view, as shown in Figure 10:

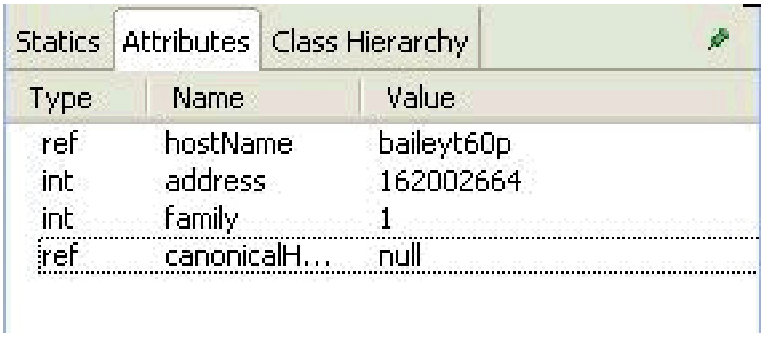
Figure 10. Inspector view for the SocksSocketImpl object



Type	Name	Value
ref	server	null
int	port	1080
ref	external_a...	null
boolean	useV4	false
ref	cmdsock	null
ref	cmdIn	null
ref	cmdOut	null
int	timeout	0
int	trafficClass	0
boolean	shut_rd	false
boolean	shut_wr	false
ref	socketInpu...	null
int	fdUseCount	1
ref	fdLock	java.lang.Object @ 0xd0374
boolean	closePending	false
int	CONNECTI...	0
int	CONNECTI...	1
int	CONNECTI...	2
int	resetState	0
ref	resetLock	java.lang.Object @ 0xd0375
ref	fd1	null
ref	anyLocalBo...	null
int	lastfd	-1
int	backlogQ	50
int	sport	0
ref	socket	java.net.Socket @ 0xd00fd0
ref	serverSocket	null
ref	fd	java.io.FileDescriptor @ 0xd0
ref	address	java.net.Inet4Address @ 0xc
int	port	100
int	localport	0

The view shown in Figure 10 gives access to the address and port fields. In this case, the port is 100, but the address field points to a java.net.Inet4Address object. Following the same process to look into the fields of the Inet4Address objects shows the results displayed in Figure 11:

Figure 11. Inspector view for the Inet4Address object



Type	Name	Value
ref	hostName	baileyt60p
int	address	162002664
int	family	1
ref	canonicalH...	null

You can see that the `hostName` is set to `baileyt60p`.

Tips and tricks

Here are a few tips and tricks that may be useful:

- Don't forget that Memory Analyzer itself may run out of memory. For the Eclipse MAT, edit `-Xmx` in the MemoryAnalyzer.ini file. For the ISA version, edit the *ISA Install*/rcp/eclipse/plugins/com.ibm.rcp.j2se.../jvm.properties file.
- If you're still running out of memory on a 32-bit version of Memory Analyzer, use the 64-bit version of Eclipse MAT or try the headless mode (see Resources). (The ISA tool currently does not currently support 64-bit.)
- Memory Analyzer writes "swap" files in the directory of the dump, which lessens the dump's reload time. These files can be zipped, sent to another machine, and placed in the same directory as the dump, making a complete reload of the dump unnecessary.
- If a dump's size does not correlate with the garbage collector at the time of the dump, consult the Unreachable Objects Histogram link in the Overview tab. The Java heap may have had a lot of garbage (for example, if a tenured collection hadn't run for some time) that Memory Analyzer removed.
- If two objects `A` and `B` do not have direct references to each other, but both have outgoing references to some set of objects `C`, then the Retained Heap of the set `C` will not be included in either of the retained sets of `A` or `B`, but rather in the retained set of the dominator of both `A` and `B`. In some situations, `B` may be temporarily observing the set `C`, which is actually the progeny of A. In this case, you can right-click on **A** and select

Java Basics > **Customized Retained Set** and use the address of `B` as the exclude (`-x`) parameter.

- You can load multiple dumps at once and compare them. Open the Histogram of the more recent dump, click the **Compare** button at the top, and choose the baseline dump.
- When you are exploring a reference tree, be aware that references can refer, directly or indirectly, back to a "parent" reference, so that you could enter an exploration loop or cycle (for example, in a linked list). Be aware of the object addresses. Also, be aware that if the class name of an object is preceded by the word `class`, that you are exploring the static instance of that class.
- The `String` value displayed in most views is limited to 1,024 characters. If you need the whole `String`, right-click on the object and select **Copy** > **Save value to file**.
- Most views have an export option, and most HTML results are created on the file system, so that data can be exported for sharing or further transformation. Relatedly, you can press Ctrl+C on any selection of rows in a grid to copy a textual representation of those rows to your clipboard.

Conclusion

Memory Analyzer was originally developed as "a fast and feature-rich Java heap analyzer that helps you find memory leaks and reduce memory consumption," as it's described on Eclipse.org. But its capabilities clearly stretch far beyond that description. In addition to their role in diagnosing "normal" memory problems, snapshot dumps can be used as an alternative to, or in addition to, other types of problem-determination techniques such as tracing and patching. Particularly with HPROF dumps and IBM system dumps, Memory Analyzer gives you memory contents such as primitives and the field names from the original source code. Using the various views covered in this article, you can explore or reverse-engineer the problem at hand, including overall footprint and memory efficiency, Eclipse bundles and classloader relationships, thread data usage and stack frame locals, exceptions, and more. OQL and the Memory Analyzer plug-in model also allow you to inspect the dump more easily using a query language and programmatic methods that can help in automating common analysis.

Resources

Learn

- Webcast: Debugging Java Applications with Memory Analyzer and the IBM Extensions for Memory Analyzer: Chris Bailey and Kevin Grigorenko lead this session on how to generate dumps and use them to examine your application: find memory leaks, inspect threads, and look at the IBM product structures using the IBM Extensions for Memory Analyzer.
- Memory Analyzer blog and Memory Analyzer wiki: Check out these resources on Eclipse.org.
- IBM on troubleshooting Java applications: Follow Chris Bailey's developerWorks blog.
- Using the -Xdump option: See this section of the IBM Java Diagnostics Guide 6 for information on `-Xdump` command-line options.
- Java HotSpot VM Options: Here you'll find the syntax for changing the location of a dump file generated with the HotSpot JVM.
- How to obtain system dumps for:
  - AIX: Enabling full AIX core files.
  - Linux: Setting up and checking your Linux environment.
  - z/OS: Obtaining SVC dumps and Setting up dumps.
  - Solaris: Troubleshooting Guide for Java SE 6 with HotSpotVM.
- `jmap`: Read the official `jmap` documentation.
- Using dump agents: Learn more about configuring the IBM dump engine to generate system dumps and heap dumps.
- Using system dumps and the dump viewer: Read about the IBM `jextract` and `jdmpview` tools.
- Automated Heap Dump Analysis: Finding Memory Leaks with One Click: Learn more about automatic leak detection with Memory Analyzer.
- How to run on 64bit VM while the native SWT are 32bit: See how to run the Eclipse Memory Analyzer Tool in 64-bit headless mode.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.
- IBM Extensions for Memory Analyzer: This IBM alphaWorks project is

developing Memory Analyzer extensions for easily analysing the state of IBM software products, including WebSphere Application Server.

Get products and technologies

- IBM Monitoring and Diagnostics Tools for Java - Memory Analyzer: Get Memory Analyzer as part of the IBM Support Assistant.
- Eclipse Memory Analyzer Tool and the IBM DTFJ Plug-in: Download the Eclipse MAT and install the DTFJ plug-in to use MAT with IBM PHD and system dumps.
- IBM Monitoring and Diagnostics Tools for Java - Health Center: You can use Health Center to request either a PHD or system dump from a running Java process.

Discuss

- Get involved in the developerWorks community.

About the authors

Chris Bailey

Chris Bailey is part of the Java Technology Center (JTC) team in IBM based at the Hursley Park Development Lab in the UK. He is the technical architect for the IBM Java service and support organization, responsible for enabling users of the IBM SDK for Java to deliver successful application deployments. Chris is also involved in gathering and assessing new requirements, the delivering of new debugging capabilities and tools, improvements in documentation, and improving the wider quality of the IBM SDK for Java.

Andrew Johnson

Andrew Johnson is a Chartered Engineer and an Advisory Software Engineer at the IBM Java Technology Center in Hursley, England. He joined IBM in 1988 after receiving a B.A. in Electronic Engineering from Cambridge University. Since 1996 he has worked on Java virtual machines, Just-in-time compilers, and tools for diagnosing Java problems. He wrote an adapter for the Eclipse Memory Analyzer to read dumps from IBM VMs and is now a committer on that project.

Kevin Grigorenko

Kevin Grigorenko is a software engineer on the WebSphere Application Server SWAT team, which provides worldwide, on-site and remote supplemental product defect support. particularly in critical customer support situations. He currently focuses on problem determination for WebSphere Application Server and related stack products, including the JVM and various operating systems. He also has a deep history in development, including Java Enterprise Edition, C, C++, Perl, PHP, Python, Ruby, and .NET.