

Monitoring Object Creation in Java Application Profiling with Rational PurifyPlus

by [Goran Begic](#)

Technical Marketing Engineer, Development Solutions
Rational Software B.V.
The Netherlands



In the [January issue](#) of The Rational Edge, I wrote about memory leaks in Java applications and possible approaches for detecting and resolving them. As I explained, contrary to common myths about the Java garbage collection mechanism's infallibility, there are objects in memory that the garbage collector cannot reach, and those objects

are called "memory leaks."

Another potential performance bottleneck -- extensive memory usage -- is much simpler to explain: While the garbage collector is cleaning memory from unused objects, the performance of the application executed through the Virtual Machine declines. Therefore, although the use of objects is at the very heart of object-oriented programming, and Java is definitely an OO language, an excessive number of objects can seriously decrease a Java application's effectiveness. For this reason, it is important to monitor the level of object creation for Java application profiling. In this article we'll take a look behind the garbage collection curtain at the world of object creation, and see how Rational Purify can help keep it under control.

Java Objects and Object Creation

There are many books written about objects in Java because Java programming is all about objects. Objects can be defined as instances of classes, which in turn can be considered templates for objects; classes define all features for a family of objects. When a new object gets created on the heap by calling `new()`, the first thing that happens is that a chunk of a memory space on the heap gets allocated. After that, the constructor for the object class is called. If the constructor is not specifically defined, then the default object will be used. Additionally, the object fields get automatically initialized, and the garbage collection marks both the object reference and the object in its list of objects and references. This means that every time we call `new()` we get more than a reference to the memory location; we actually get an initialized object.

- ▶ [subscribe](#)
- ▶ [contact us](#)
- ▶ [submit an article](#)
- ▶ [rational.com](#)
- ▶ [issue contents](#)
- ▶ [archives](#)
- ▶ [mission statement](#)
- ▶ [editorial staff](#)

A large number of small objects, especially if they are created in loops that are called often in the application, can be a major performance bottleneck. In Java (unlike in C++), you cannot allocate large memory areas simultaneously for multiple temporary objects. Instead, you must follow the same creation/destruction process for every single temporary object in memory.

Temporary local objects can also create problems. Java doesn't use the stack for such allocations in the same way that C++ does. Instead, it creates all temporary objects on the heap, but not all of them get released automatically. They are released after all the references to them are removed and the garbage collector decides to clear them. If you have a large number of objects, then the garbage collector has to be executed more often, and the program takes longer to execute.

To better understand the correlation between the number of objects created by an application and application performance, let's look at String objects.

String Objects

One characteristic that makes string objects special is their immutability. To modify a String object, you must create a new object and leave the previous one for garbage collection. This can become a problem if you repeat the operation frequently in your application.

For example, take a very simple statement like the one below:

```
public MethodA(String Expression);
```

The method takes a string object as a parameter, and we will assume that it also returns a string object, since that is very typical for interface objects. Since String objects are immutable, at least two temporary objects must be created every time the method is executed. That alone does not represent a performance issue, but it becomes an issue if you call this method several thousand -- or several hundred thousand -- times during the run. If you call the function 5,000 times, for example, then the number of created objects for such a method will be at least 10,000. Also, there will have to be 10,000 calls to the constructor of the object, and so on.

String objects are closely related to StringBuffer objects; often, StringBuffer temporary objects are invoked implicitly, as in this example of the String concatenation:

```
String str3 = str1 + str2 + str3;
```

What actually gets executed with this statement is the allocation of the new *StringBuffer* object to which the *str1*, *str2*, and *str3* will be appended and written to a new *String* when executing the *toString()* method on the new *StringBuffer* object. Please note that none of the temporary String objects is reused. After they serve their purpose, the unnecessary objects will remain in line, waiting for the mighty garbage collector to pick them up. Again, if such a String concatenation is executed in a loop, the result will be a large number of objects, more garbage collections, and a decline in performance.

Although creation of the String objects takes time, it also has advantages --

internationalization support and compatibility with existing interfaces, for example. On many occasions there is no way to avoid Strings.

Collecting Information About Object Creation

There is more than one way to monitor for excessive object creation. We will examine each option below.

Java Virtual Machine Profiling Interface (JVMPI)

The Sun Java Virtual Machines are equipped with powerful profiling interfaces through which it is possible to collect information about events and the heap usage of the JVM. Among the types of events that can be monitored through the JVMPI are:

- Byte code instruction execution
- Loading and unloading of classes
- Garbage collections
- Exceptions
- JIT method compilation
- Method calls
- Object monitoring
- Thread creation and destruction

Microsoft Java Virtual Machine doesn't support JVMPI as it is defined in the Java 2 specifications. The Microsoft solution is based on COM interfaces and callbacks. The techniques for using it are rather different than those for Sun Java JVMPI, but it can monitor all of the events listed above. The Microsoft solution also provides information about the source line execution.

Byte Code Insertion

The additional way of collecting information about the program execution is directly from the byte code. It enables Purify for Java to collect the source line execution information, for example, regardless of the Java Virtual Machine used. The technique is called Byte Code Insertion (BCI). The name resembles a Rational patented technique called Object Code Insertion (OCI), which is used with native compiled applications.

Java class files are meant to be quickly transmitted through the network to final users. In order to achieve a high level of portability, the Java code is compiled into intermediate byte code that is then interpreted by the virtual machines specific to the platforms the code is running on.

Byte code execution and its translation into machine code are not included in the Java specification. Testing tools like Rational Purify for Java add instructions to the intermediate byte code that will enable it to monitor the execution of the Java application.

The fact that all necessary symbols are included in the class files simplifies instrumentation of the byte code. The Java VM will link such instrumented code and execute it; the VM is not aware of changes in the byte code that took place between compiling and execution.

Object Creation Profiling

The JVMPI interface specifications are public, and any third party vendor can use them to gather data for their tools¹. The most rudimentary tool that takes advantage of the JVMPI is called *hprof*. It is a DLL shipped and installed with the Sun JDK. When loaded, *hprof* "listens" to JVM events and writes a log file with information about method execution times and heap usage. Unfortunately, this log file is very difficult to use; it can be extremely large because of the large number of events that JDK is processing. There are two formats for the dump file: ASCII and binary, which requires a special tool to analyze the results.

Test Application

In a book called *Java Pitfalls*, you can find the Java source file for the application that we will use to demonstrate these object creation profiling methods and tools². The References section at the end of this article contains further interesting reading material on the topic.

The test application is simple and straightforward. Its main purpose is to create a large number of objects. The main class, *Library*, creates an array of objects of the type *BookShelf*. Each *BookShelf* object creates an array of objects of the type *Book*, and every *Book* object creates an array of *TextPage* objects. After a sufficient number of iterations, the memory used for all those objects exceeds the space available for the application; the program runs out of memory and throws the appropriate exception.

Hprof can be invoked from the command line by specifying an additional option for the java executable that will load the *hprof.dll*.

```
D:\> java -classic -Xrunhprof Library
```

The resulting log file is written at the end of the run. The command line prompt includes a notification from the *hprof*: "Dumping Java heap ... allocation sites ... done."

The log file name is *java.hprof.txt* by default and is placed in the directory of the executed application. Please note that the size of the ASCII log file for this run with 1,287 objects is bigger than 4Mbytes.

Here is an example of the list of objects created on the heap as recorded with *hprof*:

```
SITES BEGIN (ordered by live bytes) Wed May 30 09:04:26 2001
```

Rank	Percent	live	alloc'ed	stack class trace name
------	---------	------	----------	---------------------------

	self	accum	bytes	objs	bytes	objs	
1	9.24%	9.24%	16388	1	16388	1	1226 [C
2	9.24%	18.47%	16388	1	16388	1	1233 [C
3	6.55%	25.03%	11628	3	11628	3	982 [C
4	4.62%	29.64%	8196	1	8196	1	1231 [B
5	4.62%	34.26%	8196	1	8196	1	1224 [B
6	4.55%	38.81%	8068	1	8068	1	985 [S
7	4.09%	42.90%	7252	3	7252	3	979 [C
8	2.52%	45.42%	4472	70	5068	85	55 [C
9	2.04%	47.45%	3612	1	3612	1	761 [L<Unknown>;
10	1.43%	48.88%	2532	1	2532	1	986 [I
11	1.42%	50.30%	2520	18	2520	18	1 java/lang/Class

The list of objects is sorted per live bytes allocated for each object, but it doesn't, for example, provide information about the memory allocated for the object and all its descendants, because they will all stay in memory as long as the observed object contains a valid reference to its descendants. Such objects are important to trace, since they may not impact the size of the allocated memory directly, but through their descendants.

In the *hprof* log file, more information about a particular object can be obtained from the trace information. It is called *trace* information because of the technique used for sampling information about the executed methods. The profiling tool, in this case *hprof*, takes regular snapshots of the call stack and calculates statistics for all the methods it finds there. For example, let's look at the object on top of the list, which is of the type *Array of Characters* (as shown in the last column with the symbol [C). The trace number 1226 with the stack trace of the methods that were called prior to the object creation gives us the following info:

```
TRACE 1226:
java/io/BufferedWriter.<init>(BufferedWriter.java:94)
java/io/BufferedWriter.<init>(BufferedWriter.java:77)
java/io/PrintStream.<init>(PrintStream.java:85)
java/lang/System.initializeSystemClass(System.java:820)
```

What we can learn from the stack trace is that this object of the type *Array of Characters* was created to buffer a stream of characters. Its size, however, didn't cause the *Out Of Memory* exception. It was the large number of books on the bookshelves that killed the execution.

There is no doubt that *hprof* offers some interesting information about the object creation, but it is extremely difficult to use. Probably the best way to use it would be to collect the information about the heap usage in the binary file and write a custom tool that would display and highlight the important information.

We can also go a step further and look at a commercial solution for Java Memory Profiling: Rational Purify for Java. Some vendors refer to memory profiling tools as "memory debuggers." In order to provide detailed and comprehensive information about memory usage, Purify for Java uses both Java 2 JVMPI and Byte Code Insertion technique.

Object Profiling with Rational Purify for Java

Purify for Java can be invoked from the command line by specifying the Rational JVMPI library instead of the *hprof* tool:

```
D:\> java -classic -XrunPureJVMPI:Purify Library
```

It tells the Java executable to load Purify for Java DLL, (*PureJVMPI.dll*) and to start the memory profiling. The option 'Purify' tells the dll to use Purify for Java as the target tool. (Specifying 'Quantify' as the target for the data collection would, instead, launch Rational Quantify, the tool for execution time analysis). The option "Classic" specifies a java executable to load the classic version of the Java Virtual Machine rather than the default, performance optimized "Hot Spot" version of the Java Virtual Machine. Future versions of Purify for Java will support the "Hot Spot" Virtual Machine from the Java Development Kit 1.3.1 onwards.

Identifying Memory Leaks

When the application starts, in the command prompt we can see the objects being created:

```
E:\Gogo\java\Library>java -classic -Xrunhprof Library
Creating BookShelf: 1, subject=Sports
Creating BookShelf: 2, subject=New Age
Creating BookShelf: 3, subject=Religion
Creating BookShelf: 4, subject=Sci-Fi
Creating BookShelf: 5, subject=Romance
Creating BookShelf: 6, subject=Do-it-yourself
```

The application continues to run, and while creating the ninth *BookShelf* object, it will exceed the maximum memory available for the application and raise an exception:

```
Exception in thread "main" java.lang.OutOfMemoryError:
```

This is a good moment to take a snapshot of the memory profile. We can use snapshots that we record during the run to examine the object creation. In my [article](#) about searching for memory leaks in the January issue of *The Rational Edge*, we saw that by comparing these snapshots, you can determine whether the Java garbage collection cleaned unused objects in memory.

This time, however, we do not need to compare two snapshots to find the memory problem. It is clearly visible within the Memory Tab of the basic Purify Data Browser window, as shown in Figure 1:

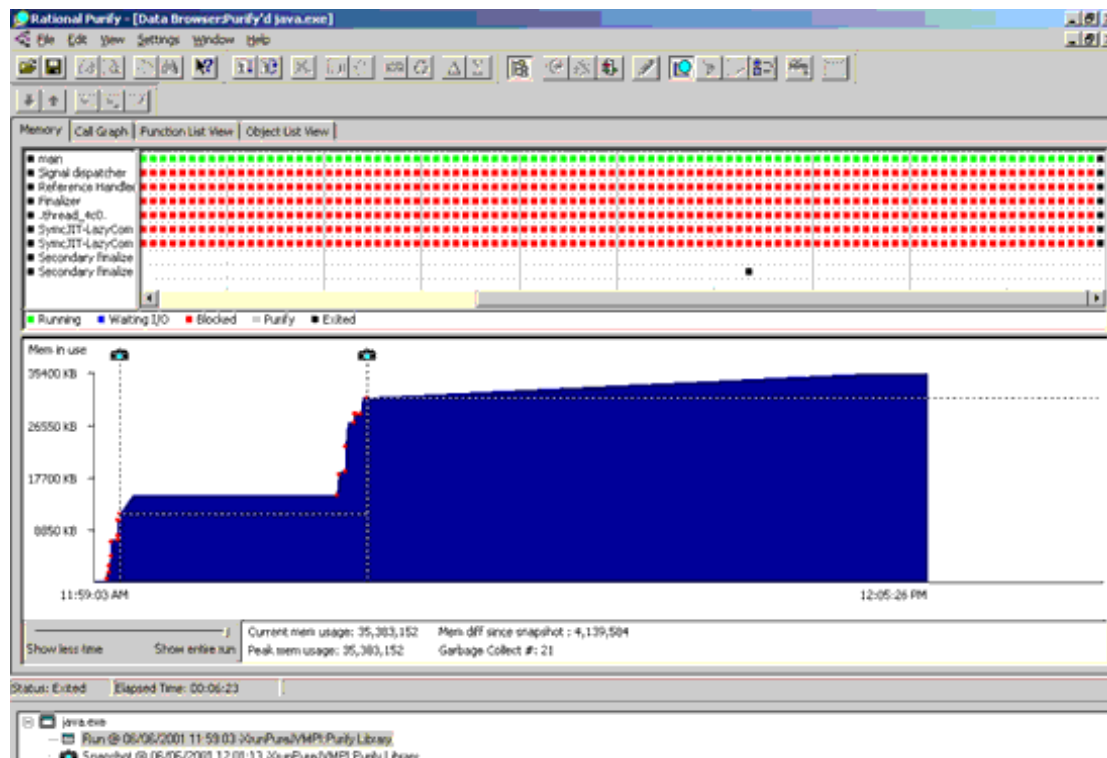


Figure 1: Rational Purify Displaying Memory In Use
Click [here](#) for full size image.

Memory usage for objects allocated for bookshelves after the sixth one -- where we took the first snapshots -- grows significantly, despite garbage collection. In this case we are not dealing with memory leaks, but rather with memory in use. All the objects in memory have valid references to them, so they are not eligible for garbage collection.

Pinpointing the Source of Excessive Memory Consumption

Another view that Purify for Java offers us is the *Call Graph*. The Call Graph highlights the application's "hot spot": the chain of calls where most of the memory was allocated, as shown in Figure 2.

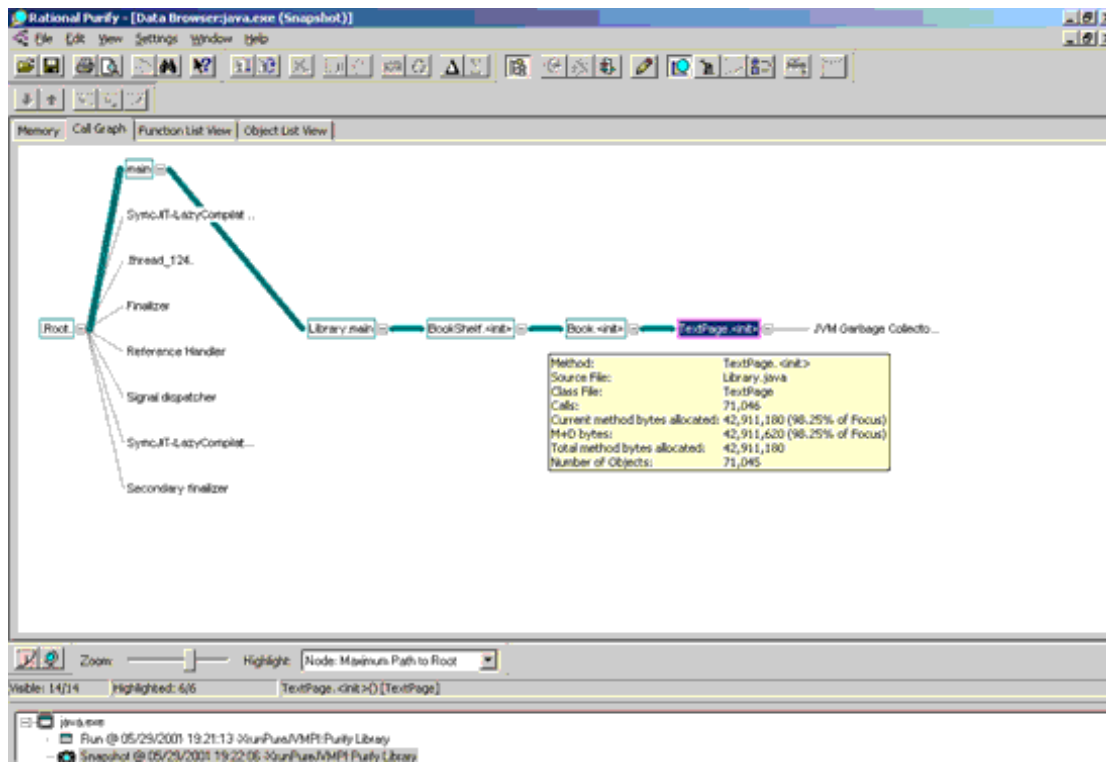


Figure 2: Rational Purify Showing the Chain of Calls with Greatest Memory Usage
 Click [here](#) for full size image.

Unlike with *hprof*, this time we are really on the way to learning more about the memory used for the numerous objects created during the run. The methods displayed on the Call Graph are constructors for instances of the class files *BookShelf*, *Book*, and *TextPage*.

This capability is new to Rational Purify for Java. As you can see, Version 2001 offers much more than just an overview of the methods where memory is allocated: It actually provides a view into Java objects and object references; it allows us to see all the objects that were "live" at the moment of the snapshot.

Finding More Information About Objects and Their References

Let's continue on the road to monitoring object creation by choosing the constructor for the Library object from the list of methods that were executed during the run, as shown in Figure 3.

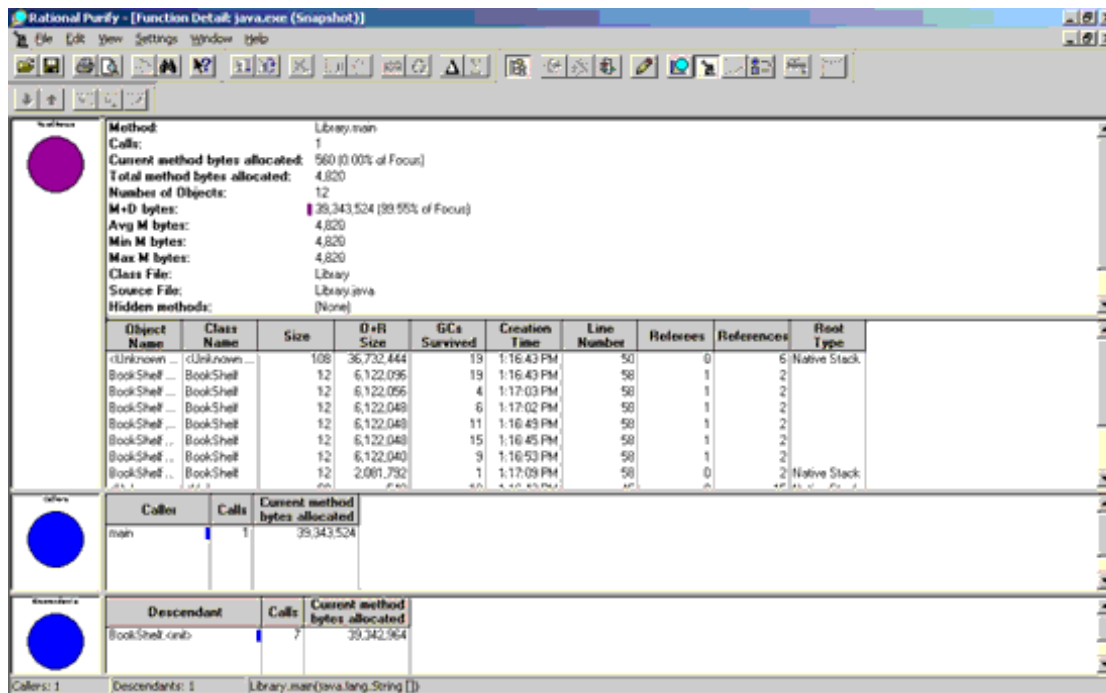


Figure 3: Rational Purify Showing Details for a Method
 Click [here](#) for full size image.

Besides information about memory allocated in the method and the list of callers and direct descendants, now there is a list of objects created in the method, along with the number of references to these objects. The list is a root for the created objects, an array of the type *BookShelf*. It stores the references to all seven *BookShelf* objects that are listed below the root.

A quick look at the source file from within the Rational Purify for Java window (Figure 4) confirms that these objects are created in the main method by calling the constructor for the new *BookShelf* object.

```
class Library
{
    public static void main (String args[])
    {
        String subjects[] = { "Sports", "New Age", "Religion",
            "Sci-Fi", "Romance", "Do-it-yourself",
            "Cooking", "Gardening", "Travel",
            "Mystery", "Fantasy", "Computers",
            "Business", "Young readers", "JW Books" };
        BookShelf[] shelves = new BookShelf[25];

        try
        {
            for (int i = 0; i<shelves.length; i++)
            {
                String subject = subjects[i];
                System.out.println ("Creating BookShelf: " + (i+1) + ", subject=" + subject);
                shelves[i] = new BookShelf (subject);
            }
        }
        catch (OutOfMemoryError e)
    }
}
```

Figure 4: Source Code Line for Creating a New Bookshelf Object

The *BookShelf* objects array is also easy to recognize (Figure 5). An array in Java is a special object that keeps references to the actual objects of the type for which the array is created.

```

class Library
{
    public static void main (String args[])
    {
        String subjects[] = { "Sports", "New Age", "Religion",
            "Sci-Fi", "Romance", "Do-it-yourself",
            "Cooking", "Gardening", "Travel",
            "Mystery", "Fantasy", "Computers",
            "Business", "Young readers", "JV Books" };
        BookShelf[] shelves = new BookShelf(25);

        try
        {
            for (int i = 0; i<shelves.length; i++)
            {
                String subject = subjects[i];
                System.out.println ("Creating BookShelf: " + (i+1) + ", subject=" + subject);
                shelves[i] = new BookShelf (subject);
            }
        }
    }
}

```

Figure 5: *BookShelf* Objects Array in Source File

If we open the *Unknown* array object and look at it in the Object Detail window (Figure 6), then the story about the array object and the references to the real object becomes very clear:

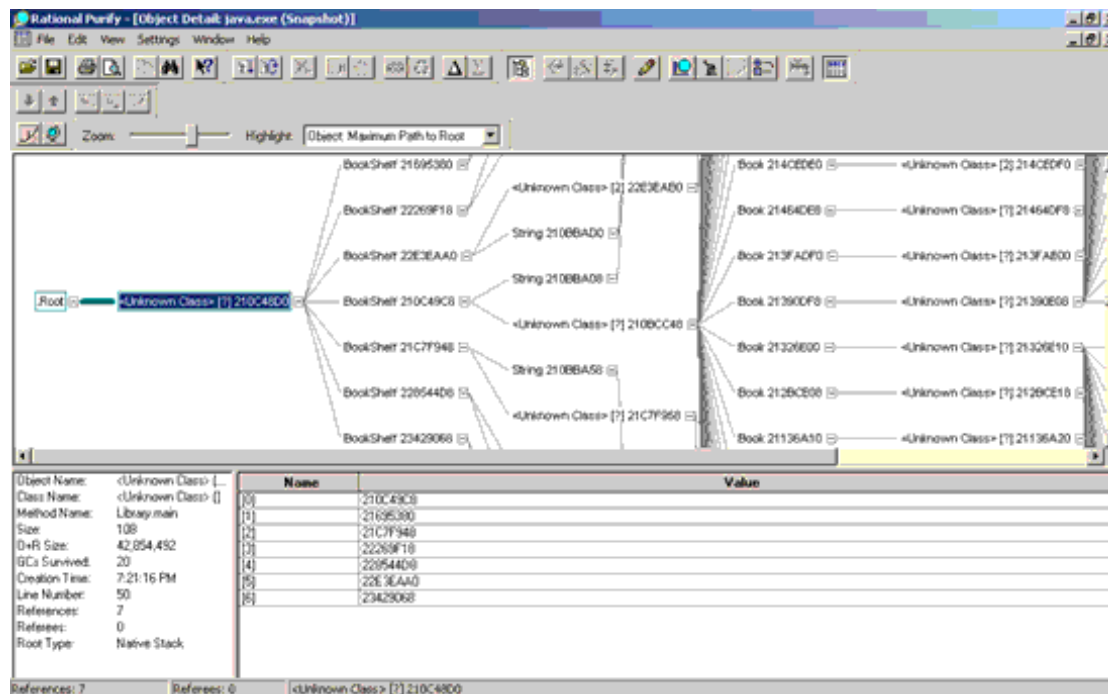


Figure 6: Detail View of an Array Object
Click [here](#) for full size image.

The lower right portion of the screen in Figure 6 shows the Object Data Window, with references to the array of *Books*. Each *BookShelf* object contains the reference to the array of *Book* objects. Rational Purify for Java displays only one reference per array element to keep the overview clean.

If you drill down into the *Unknown* class -- which is actually an array of *Book* objects -- then you can see all the references and relationships among the objects in the Object Data Window. Then, double-clicking on a reference will lead you directly to the object behind that reference. If we choose for example, the third element of the *BookShelf* array (referenced as 21C7F948), then we will see that it has two references. One is a reference to the array of *Books* (21C7F958), and the other is a reference to the *String* object. If we continue exploring the latter reference, we will see that it is a character array with the content *Religion*, and that is indeed the third *BookShelf*.

There are numerous ways to exploit the capabilities of a commercial memory-profiling tool like Purify for Java. One of them would be to monitor the size of the largest objects and descendant groups of objects by taking several snapshots throughout the run of the application. You could then identify the objects that consume the most memory, as well as the numbers of certain objects.

Optimizing Memory Usage in Java Applications

We have seen how excessive object creation in Java can cause problems and how a memory profiling tool such as Rational Purify for Java can help programmers optimize memory usage in Java applications. Memory profiling tools can also measure and document the effects of code changes on an application's overall performance, and the references listed below provide many suggestions and solutions for improving application performance. By following these suggestions and using the right tools throughout development, programmers can help ensure greater success for Java projects.

Footnotes

¹Java Virtual Machine Profiler Interface specification (JVMPi):
<http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/jvmpi.html>

²Michael C. Daconta (Editor), Eric Monk, J. Paul Keller, Bohnenberger, Keith Bohnenberger, "Java Pitfalls", John Wiley & Sons, April 2000.

References

1. Memory Profiling in Java
http://www.therationaledge.com/content/jan_01/m_memjava_gb.html
2. Java Virtual Machine Profiler Interface specification (JVMPi):
<http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/jvmpi.html>
3. JavaSoft HAT:
<http://developer.java.sun.com/developer/onlineTraining/Programming/JDCBook/perf3.html#profile>
4. Rational Developer Tool documentation:
www.rational.com/products/pqc/
5. Michael C. Daconta (Editor), Eric Monk, J. Paul Keller, Bohnenberger, Keith Bohnenberger, "Java Pitfalls", John Wiley & Sons, April 2000.

6. Steve Wilson and Jeff Kesselman, "*JAVA Platform Performance, Strategies and Tactics.*" Sun, 2000.

7. Jack Shirazi, "*JAVA Performance Tuning*". O'Reilly, 2000.

8. Craig Larman and Rhett Guthrie, *JAVA 2, Performance and Idiom Guide.* Prentice Hall, 2000.

9. Patrick Niemeyer and Jonathan Knudsen, "*Learning Java,*" O'Reilly, 2000.

10. Peter van der Linden, "*Just JAVA.*" SunSoft, 1996.



For more information on the products or services discussed in this article, please click [here](#) and follow the instructions provided. Thank you!