TSS Editors Note:
Manning Publications and TheServerSide.com have partnered to bring you this chapter of Java Doctor, a new book, in which you can contribute to and get published! To contribute, read the tips chapter 9 (available on TSS) and send new tips to javadoctor@manning.com.

Profiling

In this Chapter

- **Introducing profilers and the values they can provide**
- **Profiling approaches: JVMPI/TI and bytecode instrumentation**
- **Profiling memory mismanagement with HPROF and Optimizeit**
- **Profiling thread contentions with HPROF and Optimizeit**
- **Profiling CPU usage with HPROF and Optimizeit**

Many of us have seen TV shows where a gifted individual arrives at a crime scene, touches a piece of cloth or hair, and begins to visualize what took place earlier. Known in the crime fighting industry as a *profiler*, such an individual's responsibility is to determine the characteristics and behavior of the entity or entities that may have perpetrated a crime. Often this requires looking at scant pieces of evidence, possibly listening to testimony from individuals affected by the incident, and digging into historic cases that seem to have similar traits. It's a critical job, and profilers have been instrumental in solving many cases.

In the world of Java, profilers perform a strikingly similar role. The major difference is that our profilers are smart programs instead of smart individuals. And, of course, profilers are used to identify root-cause problems in applications rather than criminals.

In this chapter, we'll turn our attention to observing an application in a more, well, prolific sense. This becomes necessary when problem-solving using methods as described in previous chapters isn't sufficient or is too isolated. If starting from the lowest stack (the operating system) and moving up to the JVM doesn't point to the source of a systemic problem, then you may have to take a deeper look into the application via a profiler.

The objective of this chapter is to wet your feet and whet your appetite for profiling. In fact, profiling and the technologies surrounding it could be the subject of its own book. Therefore, we'll look at simple pieces of source code and run them on profilers to illustrate the value proposition to you. The goal is not to promote any single profiler over others, but rather to let you decide what profilers can do for you.

5.1 What is profiling, and what are profilers?

No, we're not talking about hiring a psychoanalyst to sit down, touch a server, and then draw a sketch of the perpetrating code (but if you know someone who can do that, please let us know). Profilers are tools that let you take an expansive and more holistic picture of your application during run-time. They

can show CPU consumption, memory profile, lock contention, and garbage-collector activity in one fell swoop.

But, you may ask, haven't we already talked about this information being provided by the JVM, in the previous chapter? After all, if you need to see whether there is a thread deadlock, you can do so by sending a thread dump signal to the JVM (see chapter 4), right? The answer is yes and no. One of the strengths of a profiler is its ability to leverage information provided by the JVM and provide a comprehensive and a more correlated analysis. This means that a profiler should be able to take all the information a JVM spews out, analyze it, and provide a more insightful conclusion or representation of activity. Given that profilers try to give you a picture of the complex inner plumbing of your application, it's no surprise that many commercial profilers tend to have a visual interface that is easy to understand, especially if you know what you're looking for.

Recall from chapter 4 that JVMs have API hooks, such as the JVMPI, which profilers can use to trap events and obtain fine-grained information about the JVM's doings. Profilers may be used early, as the first step in troubleshooting, or after exhausting several troubleshooting strategies. Different bugs and systemic problems may manifest themselves with similar symptoms, so only certain issues are visible with profilers; other problems require tools from the other stacks, such as the JVM or operating system. When you're following a bottom-up approach, you'll use profilers during the later stages, after the lower stacks have been analyzed. In the top-down approach, on the other hand, profilers will be used early on, either from the onset or right after a code review

You can use profilers when you see or sense problems with scalability, reliability, or performance. Applications may suffer these quality problems when unnecessary CPU *hot spots* exist (sections of code that consume large amounts of CPU). In the case where there is excessive object creation and retention, which may result in high garbage-collector activity, a profiler can help identify memory-allocation sites. Another source of systemic problems can be *thread contention*. Profilers can identify this by visually showing threads that are competing for locks when the application is running. In addition to identifying such problems, we'll also walk you through examples of them in the coming sections.

Profilers gather data for analyses in two fundamentally different categories. Some are based on the profiling interface of a JVM (for example, JVMPI/JVMTI) from which profiling information is retrieved, and others are based on modifying the bytecode for measuring various metrics. Profilers based on the profiling interface of the JVM are more detail oriented and are generally more helpful when a problem is reproducible in a controlled environment. Bytecode instrumentation-based profilers are more helpful when you're attempting to profile applications in production. We'll look more deeply at these two types of profilers later in this chapter.

Note

Typically, profilers aren't used for bugs related to the functionality or business logic of the application, but rather are used for systemic behaviors. For instance, a Java class that isn't calculating the total sum of an order correctly is an example of a functional or use-case issue. As such, bug identification should be done through a debugger.

5.2 Obtaining a profiler

Naturally, for profilers to be of any use, you need to know how to best leverage them to gain a better understanding of an application. An application has many traits, and our goal is to identify the traits that lead to problems and group them into categories, such as heavy memory allocation sites, memory leaks,

CPU hot spots, unnecessary object retention, and thread contention. These types of issues are the usual culprits of performance and scalability downturns and are thus our primary focus. This chapter will explain each of these categories in later sections.

But first, how and where do you get a profiler? Tons of them are available. Table 5.1 lists some of the better-known commercial profilers, and table 5.2 lists open-source or free profilers. Of course, your needs may be met by the profilers that come bundled with the Sun JVM, such as HPROF, or by the `-Xprof` and `-Xaprof` JVM switches. We suggest that you download the trial editions of these profilers (if provided) and tinker with them to see which is more suitable for your needs. Just like any other type of Java tool, each product has pros and cons that you should consider and evaluate before purchasing.

Table 5.1 Commercial profiler

Profiler	Vendor	Website
JProfiler	ej-technologies	www.ej-technologies.com
JProbe	Quest Software	www.quest.com
DevPartner	Compuware	www.compuware.com
Optimizeit	Borland	www.borland.com
Indepth for J2EE	VERITAS	www.veritas.com
Introscope	Wily	www.wilytech.com
PerformaSure	Quest Software	www.quest.com

Table 5.2 Free or open-source profilers

Profiler	Source
xdProf	http://xdprof.sourceforge.net/
*J	www.sable.mcgill.ca/~bdufou1/starj/
JMemProf	http://oss.metaparadigm.com/jmemprof/
JRat	http://jrat.sourceforge.net/
JMP	www.khelekore.org/jmp/
GCSpy (see appendix E)	www.experimentalstuff.com/Technologies/GCspy/
HPROF	Sun JVMs

For our purposes, this chapter will primarily use a pair of profilers to obtain observations and demonstrate general profiling capabilities and virtues. The first tool is Optimizeit from Borland Corporation. This definitely isn't a marketing plug for Borland—we don't endorse or specifically recommend this or any product. However, this tool has proven to be popular in the Java community, and we find it useful for learning and providing nice screen snapshots. The second tool we'll us is HPROF, which comes free with any post-1.2.x Sun JVM. Similar profiling tools should be available with the IBM JVM (HPROF or JPROF). In addition to these two profilers, we'll also interject the flow by mixing it up and dabbling with some of the open-source offerings.

For those who like free things (like us), using HPROF and PerfAnal is a good way to get familiar with JVMPI and profiling. HPROF doesn't have an agent, because it hooks directly into the JVMPI of the HotSpot JVM. Since it isn't an agent with a wire protocol but an output file, other tools can use the output file and give a more visual presentation. For example, PerfAnal is a tool that feeds off HPROF. You can download PerfAnal and associated documentation freely from the Sun Developer website (developer.sun.com). We'll look at HPROF and PerfAnal later in the chapter.

5.3 Understanding how profilers work

There are two types of profilers: those that are based on an interface available from within a JVM, and those that inject bytecode into existing Java class files. Each has advantage and disadvantage, ranging

from overhead to the amount and depth of information provided. The following sections highlight these differences and their implications.

5.3.1 JVMPI/JVMTI/agent-based profilers

Chapter 4 was filled with information about how to use the JVM to understand the behavior of the application. In this section, we talk about how profilers take advantage of the JVM by using an interface exposed by it to display detailed information about the application being executed. To enable this behavior, you must tell a JVM via a switch that it needs to enable the profiling interface (by default, it's turned off and inaccessible). If started, the JVM can then send events to the profiler each time the application code triggers certain conditions, such as object creation, method invocation, or any number of events that we'll discuss shortly.

The JVM Profiling Interface (JVMPI) existed in JVMs 1.4 and earlier, and the JVM Tool Interface (JVMTI) replaces JVMPI in 1.5. In this book we mainly focus on JVMPI, because as we're writing this chapter, 1.5 has just been released. However, we know that JVMTI is a more robust interface with lesser overhead. The JVMTI has a more granular approach to profiling, whereby the application is affected only by overhead in keeping with the task being profiled. This wasn't the case in JVMPI, where the overhead was significant whether the CPU usage or the thread contention was being profiled. This change in interface should concern you only if you're a developer of profiling technology and thus interact with these interfaces. For users of profilers such as HPROF or Optimizeit, this change should have minimal impact, because it's the responsibility of the vendor (such as Borland) to interoperate with the new JVMTI API.

Figure 5.1 shows how a profiler like Optimizeit traps events and requests information from the JVM. The JVM is accessed through a known interface, such as a JVM Profiler Interface, by an *agent*. The agent then communicates the gathered information to a profiler running remotely through a network connection, and finally the profiler front-end displays the given data in a more coherent and easy-to-understand manner.

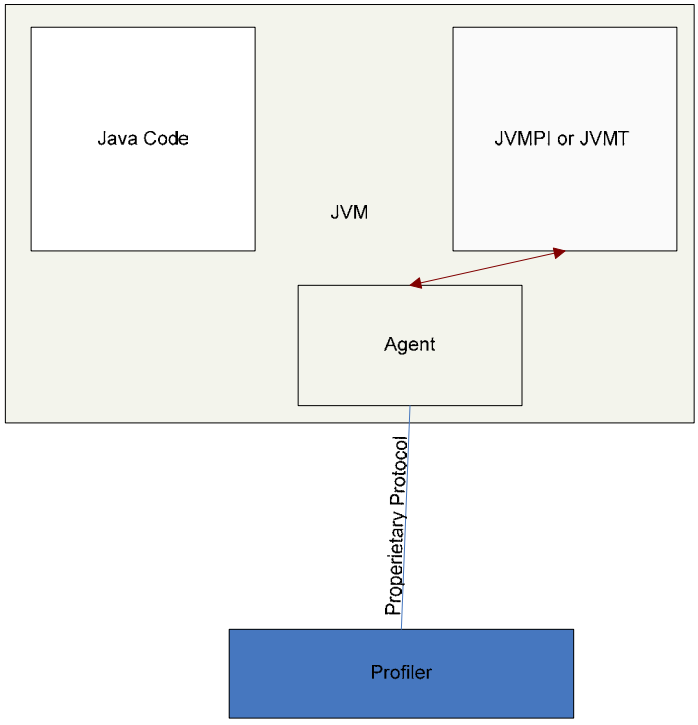

Figure 5.1 The interaction between a JVM and a profiler agent, as used by JVMPI/JVMTI/agent-based profilers

For the profiler to connect to the JVM, most profiler tools provide an option that lets Java applications be invoked via the profiler instead of your command line. That is, you can use a profiler's front-end directly to start your application with the appropriate profiling switches automatically turned on in the JVM. Another way for profilers to profile an application is to connect to an independently executing JVM using a profiler agent. The benefit of the profilers that use the agent compared to those that invoke the JVM independently without an agent is that having a profiler run remotely on a separate machine allows for faster execution of the application, because profilers require plenty of CPU cycles. Another advantage is that you can sometimes profile certain applications that run on a JVM wrapped by another native application or a script, by passing the JVM the required parameters.

Both means of connecting to the JVM are intrusive in nature and thus usually change the application's behavior. The effect may not be a showstopper, and some applications may even continue to display the same characteristics as without a profiler, but being aware of the intrusion's impact is critical to successful troubleshooting. Since JVMPI-based profilers are intrusive and therefore measurably affect the performance of the application, it's highly recommended that these profilers not be used in production environments. If it's absolutely necessary to use a profiler in production (you should do so only as a last resort), try to make sure there is adequate load-balancing capability in place to allow for most user requests to be routed to other application instances.

5.3.2 Bytecode instrumentation

Some profilers don't use an API that is embedded in the JVM but instead use bytecode instrumentation. With this approach, a profiler takes a compiled Java class and then injects or modifies it with additional proprietary bytecode. Most of this additional code is logic that aids in calculating or gathering statistics about that class's activities. This means of profiling isn't new and has been used in native languages such as C++, where the runtime execution had no virtual machine for retrieval of such information. For a Java developer, it would be akin to inserting `System.out.println`s or `log` statements (which may have a performance impact due to I/O) during development time to trace activity.

Of course, code injected by profilers uses more sophisticated techniques and methods and can automatically remove what was injected; the developer need not manually remove injected statements. Also, consideration needs to be made for some intelligence on the part of the instrumentation profiler and the user, to determine what classes and what regions of a particular class should be modified. A wholesale bytecode instrumentation of all the classes in an application can result in an overload of information output. Because of this, most bytecode instrumentation–based profilers require you to specify target class names. Selective instrumentation of class bytecode reduces the overhead compared with the instrumentation of all methods in all the classes, due to the work involved in generating and gathering of data.

The advantage of the bytecode instrumentation approach is that performance isn't hit as hard as via the JVMPI. The modified code runs within the JVM and in tandem with the application, as opposed to causing numerous event generations from the JVMPI. The downside is that after every recompile of your application, you need to instruct the profiler to instrument the newly compiled bytecode. However, with tools such as Ant to help automate such tasks, the downside isn't so bad. Table 5.3 gives a breakdown of the advantages and disadvantages of the different approaches used for profiling an application.

Table 5.3 Comparison of techniques and APIs used by profilers

Method	Pros	Cons
JVMPI	Lots of fine-grained information	Overhead of layers
Bytecode instrumentation	Better performance Minimal impact on application behavior	No GC information Limited types of information

Manual (logging, `System.out.println` insertions)	Very targeted to what information you need Lower impact on performance	Limited types of information Maintenance of code (removal, and so on) Performance hit if using IO
JVMTI	Standardized interface Fine-grained information Overhead determined by granularity of information retrieved	Relatively new technology (1.5 and beyond)

Note

Bytecode instrumentation–based profilers can be used in production with limited caution. Commercial tools that use bytecode instrumentation eagerly tout this benefit. However, you need to use good judgment when deciding which methods and classes to instrument, because the more extensive the instrumentation, the larger the overhead.

5.3.3 Right tool, wrong conclusions

Both kinds of profilers—JVMPI-based and, to a lesser degree, bytecode instrumentation—can incur a large overhead, which can slow the application significantly. The slowdown can cause the application to change its dynamic behavior, resulting in known systemic bugs disappearing while newer ones spring up. This can be frustrating when you're trying to diagnose a systemic problem through a profiler, which instead of showing the true cause, points to a profiler-induced issue.

As an example, let's take a case of profiling an application that does extensive database querying. Generally, when you use profilers on such application, an abnormally large amount of time is shown to be spent in accessing the database. In reality, database access may not be a bottleneck, but it will appear that way because of the way event generation within JVMPI hampers JDBC-based communication. In the absence of any quick test to determine if the noticeable issue in a profiler is an anomaly, it's recommended that you take time in connecting the dots between a known symptom and a noticeable issue. If the noticeable issue seems completely out of sync with the perceived problem, than the priority of digging deeper into it should be reduced. This involves investigating all possible issues made apparent through the use of a profiler, keeping an eye out for possible anomalies.

A more important lesson is to never celebrate a troubleshooting victory prematurely. When you're identifying the source of a systemic bug, more often than not the application is plagued with multiple problems that all need to be resolved before you can claim success.

Now, let's get on to the meaty stuff: using a profiler to identify and address memory, CPU hotspots, and other issues.

5.4 Memory mismanagement

The first area we'll look at is memory mismanagement, commonly referred to in the industry as *memory leaks* (we examined some memory issues in chapter 3). In this scenario, objects that are deemed to be garbage by the Java developer aren't collected by the garbage collector. In fact, the garbage collector passes over these objects as if they're legitimately existing ones because they're still being unnecessarily referenced. In this section, we'll show how to identify such leaks through the use of a profiler. We'll do this walking you through a complete cycle of how to handle these problems from their onset.

The first major indication that a memory leak exists is the occurrence of the `java.lang.OutOfMemoryError`, which may also occur if the logic of the application requires more heap space. Getting an `OutOfMemoryError` early in the execution of an application can offer certain advantages in comparison to memory errors thrown over long durations. Memory errors that are thrown after a long period of execution are usually induced as small fragmentary memory leaks accumulate. When an application dies within minutes of starting because of memory constraints, the probability of quickly locating the leak is potentially higher because the leak grows large rapidly in short time span. Unless the JVM has been allocated a disproportionately low amount of maximum memory (`-mx`), the code responsible for the spike is likely to be solitary. In the case of long-running applications inducing memory errors, it's possible that the minor leaks come from multiple, dispersed code sources. If so, they will be difficult to trace.

Another disadvantage of leaks occurring over long spans of time is the time itself. Reproducing the error can become impossible within a reasonable timeframe. Of course, you can't control when the application throws a memory error. The point is that these covert, enduring leaks should not be ignored. You can discourage long-running applications from slowly leaking by involving a profiler. Profilers should be involved early in application development to ensure that such issues are identified as soon as possible—especially and even when everything seems to be running smoothly.

If you're receiving an `OutOfMemoryError`, or if you have a gut feeling that your application may have leaks, or if your application's throughput goes through a gradual slowdown as time progresses (based on increase of garbage collection frequency due to reduction of available memory), then one of the first things you can do is to run the application with the `-verbose:gc` flag. As discussed in chapter 4, `verbose:gc` generates information about the garbage collector's activities to the console. Using `-verbose:gc` can be a quick and easy way to get information about the garbage collector's activity prior to loading a heavy-duty profiler. If you start to notice a pattern where fewer objects are claimed by the garbage collector for each successive collection, then you can be sure that a leak exists. To isolate the problem and drill down further, use a profiler.

Before we delve deeper into profilers and their ability to track down memory leaks, let's examine the behavior of the objects involved in a memory leak. Any object that has completed its useful life but is unable to be garbage-collected is a participant in a memory leak. Applications achieve a state of equilibrium once they're done with the initializations involved during startup. As a result of this equilibrium, objects of classes follow a cyclical pattern of increasing in number before being collected by the garbage collector.

After each collection, the number of objects for a given class returns to a lower count before increasing again as the application executes. Much like the human body, the application is *homeostatic* (the tendency of the body to adapt and change in order to eventually reach biological equilibrium). Keeping this in mind, you can be fairly certain that objects of classes involved in memory leaks deviate from this pattern, In other words, the object count *never* returns to an initial equilibrium count. The object count for an object that is a participant to a memory like will, after each collection, have a pattern of becoming larger and larger. If you can identify the class and object-creation place for those objects that continue to increase in count despite garbage collections, then you've found your leak!

5.4.1 Identifying leaks through Optimizeit

A common cause of memory mismanagement is inserting objects in a collection and then forgetting they're there, so they're never removed. In such a scenario, even though the developer may consider the inserted objects to be garbage, the JVM is unable to collect them because they're still referenced by the `Collection` class. We'll present a sample application, and then use a profiler to show how such leaks can be identified in more complex applications.

In the `MemoryLeak` class (listing 5.1), there exists a collection into which we continuously insert objects throughout the life of the application. During the execution of the application, with `-verbose:gc` as a JVM parameter, the behavior of more and more instances of the same object not being garbage-collected becomes apparent. In the class, we insert the `StringBuffer` objects into the collection, making the assumption that we won't be referencing them from the collection at a later time. As developers, we assume they're no longer useful to us and are thus garbage. The resulting memory leak causes a portion of the heap to become uselessly occupied and, after a certain point in time, stalls the application altogether. How can we identify this problem with a profiler?

Listing 5.1 The `MemoryLeak` class, which showcases memory mismanagement

```
import java.util.*;

public class MemoryLeak {
    public static int MAX_ITR=70000000;
    public static int STR_SIZE=100;
    protected static List coll = new ArrayList();

    public static void main(String[] args)throws Exception {
        StringBuffer foo;
        for(int i=0;i<MAX_ITR;i++){
                foo=new StringBuffer(STR_SIZE);                    #1
                if(i%100==0) {                                    |#2
                   coll.add(foo);                                 |#2

                  /* Do some work with foo */

                  foo = null;                                      #3

//Thread put to sleep to slow down execution of program.
                   Thread.currentThread().sleep(2);
                }

               // if(i%10000==0) coll.clear();                     #4
        }
    }
}
```

(annotation)<#1 Create new object of type StringBuffer
(annotation)<#2 Add every 100th object to a collection
(annotation)<#3 Nullify foo, forgetting that ArrayList holds separate reference
(annotation)<#4 Uncomment line to fix memory leak

In the `MemoryLeak` class, while most of the instantiated objects quickly become garbage (as is the behavior with most applications), our application manages to retain some of these objects for the duration of its life. As we continue to add objects to the collection for the life of the application, the JVM's heap start to fill up; at some point, the garbage-collection activity dwarfs everything else. The final nail in the coffin is an `OutOfMemoryError`. As you can see, not all of the instantiated objects are placed in the collection because this helps to show how the heap usage of a typical application looks.

Now, we take the code and run it through Optimizeit. We do this by invoking the Java application from within Optimizeit through its option of executing a program. We use the memory profiling option once our application starts executing. In the memory profiling view, Optimizeit shows us line charts of object counts for each class. Remember, to identify memory mismanagement and leaks in an application, it should be allowed to execute until it gets to a point when all/most of the initializations have completed.

Generally, startup and shutdown are aberrations in the behavior of the normal dynamics of an application. For certain applications, it's difficult to assess exactly when the initializations have completed and the normal execution of the application has begun, but don't worry: If it's difficult to assess when most initializations have completed, let the application run for a sufficient amount of time to reduce any initialization-related noise that may muddy the profiling metrics. For example, in a Swing application, you may want to wait till the entire GUI is loaded, whereas for a GUI-less application, you should wait 10 minutes or until CPU consumption is down. To accomplish this, either ignore the profilers' monitors and other output until the application reaches the equilibrium state, or start the application outside of the profiler (in this case, Optimizeit) and connect it after sufficient time has elapsed for the application to reach its post-initialization state.

Figure 5.2 shows a screen capture of Optimizeit taken right after the sample application code was started. For the example, we haven't put in any initializations for the sake of simplicity, and hence we didn't have to wait for the initialization to complete. Note the upper-left frame: The darker line (a red line on a colored display) signifies the maximum heap size, and the lighter (green) line signifies the portion of heap currently in use. As the figure shows, the heap continues to grow and shows no sign of reaching equilibrium. Once you notice such a graph in a post-initialization phase of an application that has been running for some time, you can conclude that there is a high probability of some type of memory mismanagement in the Java application. Of course, this conclusion is based on the assumption that soft/phantom references aren't being used in the application, because their use will make it difficult to tell between memory mismanagement and general heap-sizing problems. (For discussion of reference types, see chapter 3.)

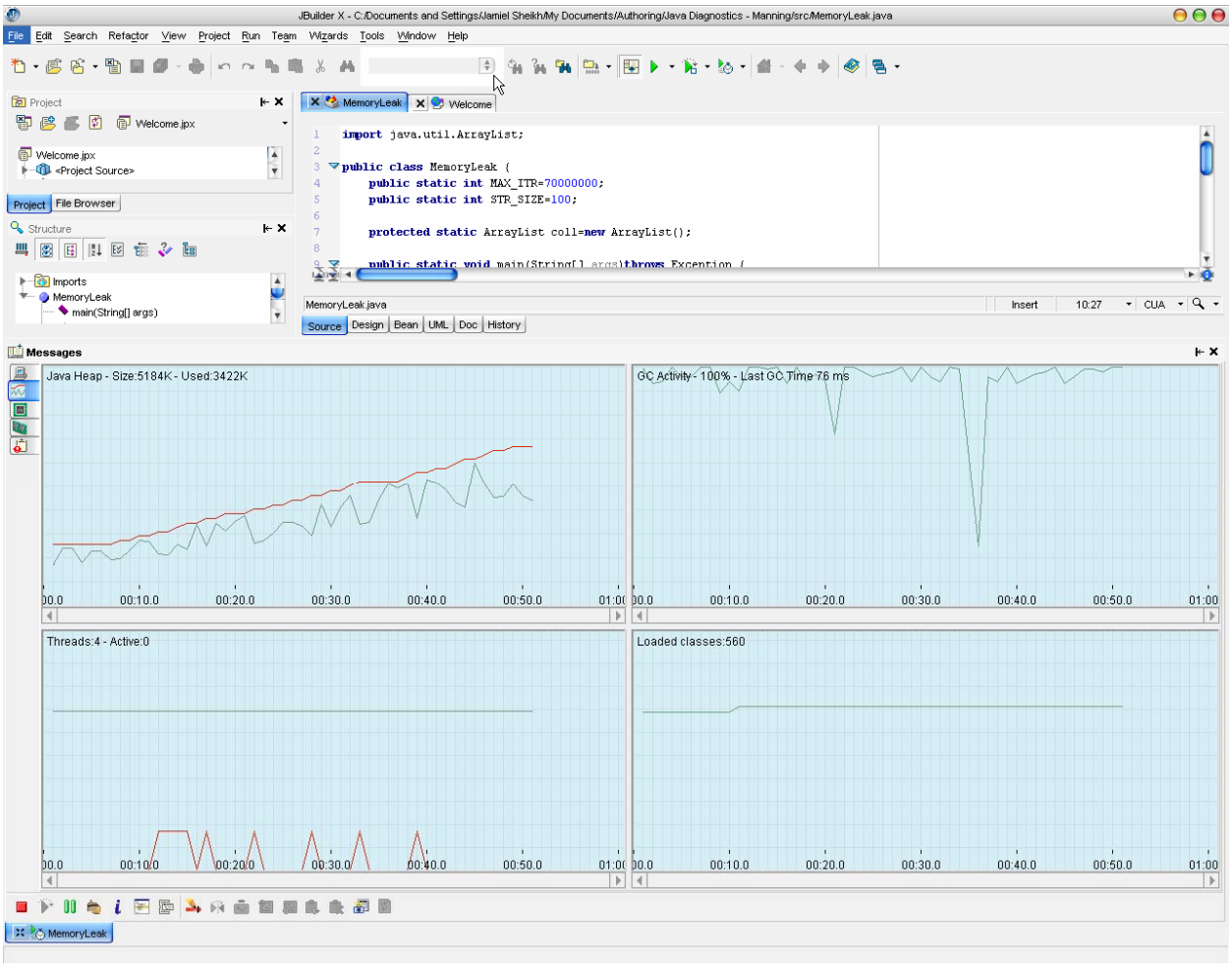

Figure 5.2 Memory profiling in Optimizeit (integrated with JBuilder X here) consists of four frames. The upper-left frame denotes the total heap (dark line) and the heap currently in use (light line). The upper-right window shows garbage-collection activity. The lower-right frame gives the number of Java classes loaded, and the lower-left frame gives the number of the threads in the application being profiled. If you're profiling memory, the upper-left window shows how the heap fails to reach equilibrium and continues to grow unchecked.

With a strong hunch about a memory leak, we next attempt to identify the class of the objects that partake in the leak. To do that, we use the *heap view* in Optimizeit (see figure 5.3) and try to locate the objects that keep increasing in count without coming back to an equilibrium state. The heap view allows us to drill down and see what objects currently exist in the heap. As the heap occupancy decreases after each garbage collection, with all well-behaved objects returning to an equilibrium count, it's a matter of identifying the odd class out.

An easy way to do this is to identify objects with a high instance count in the heap view, because a memory leak is the result of many objects failing to be garbage collected. Optimizeit has a marker feature you can use to mark the current state of the components in the JVM, including the count of objects. Use the marker, and then view the heap state during execution for objects that continue to increase without returning to the marked state. All such objects become candidates for further investigation.

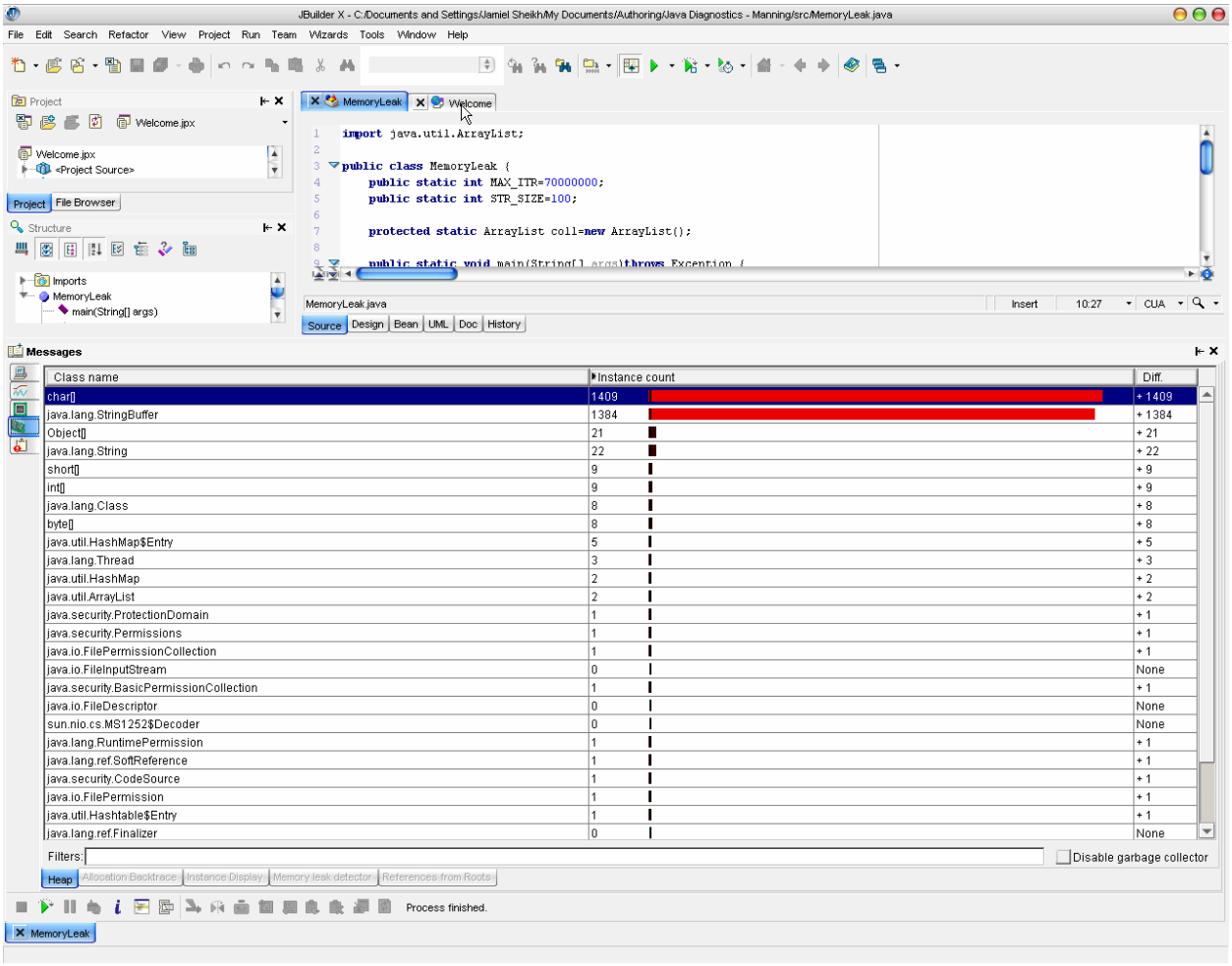

Figure 5.3 The Optimizeit heap view with instance counts for different classes

In figure 5.3, you can see that four classes have object counts with the largest difference from the marked state:

- An array of `char`
- `java.lang.StringBuffer`
- An array of `Object`
- `java.lang.String`

The black lines in the middle of the horizontal bars depict the marked state of the object count for the classes and serve as a rough point of reference for an equilibrium. You can see that total object counts of `StringBuffer` and `char` arrays are high, as well as their difference from our marked equilibrium state.

Given that the high instance count of `char` arrays is coupled with the high instance count of `StringBuffer`, we need a way to drill down further. We can then try to determine the source of `char` array creation by performing a *backtrace*, which shows what code is responsible for what instantiations of a particular object type. Using the backtrace in Optimizeit leads us to the `StringBuffer` class. Listing 5.2 shows the implementation of `StringBuffer`, which uses an array of `char` internally for storage. We therefore move our attention to the `StringBuffer` objects, instead of the array of `char`, because they're internally created to constitute a `StringBuffer`. Applying a backtrace to the `StringBuffer` identifies that the `main()` method is where 100% of the `StringBuffer` instantiations take place, as shown in

figure 5.4. In a more realistic situation, the backtrace would show multiple methods and would be slightly harder to figure out. However, the strategy remains the same; you focus on the code with a high percentage of instantiations and the greatest difference in count compared with equilibrium.

Listing 5.2 `String`s and `StringBuffer`s store their values as array of characters.

```
/*
 * @(#)StringBuffer.java      1.78 03/05/16
 *
 * Copyright 2003 Sun Microsystems, Inc. All rights reserved.
 * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 */

package java.lang;

[Some text deleted]

* @author      Arthur van Hoff
 * @version       1.78, 05/16/03
 * @see     java.io.ByteArrayOutputStream
 * @see     java.lang.String
 * @since   JDK1.0
 */
public final class StringBuffer
    implements java.io.Serializable, CharSequence
{
    /**
     * The value is used for character storage.
     *
     * @serial
     */
    private char value[];
```

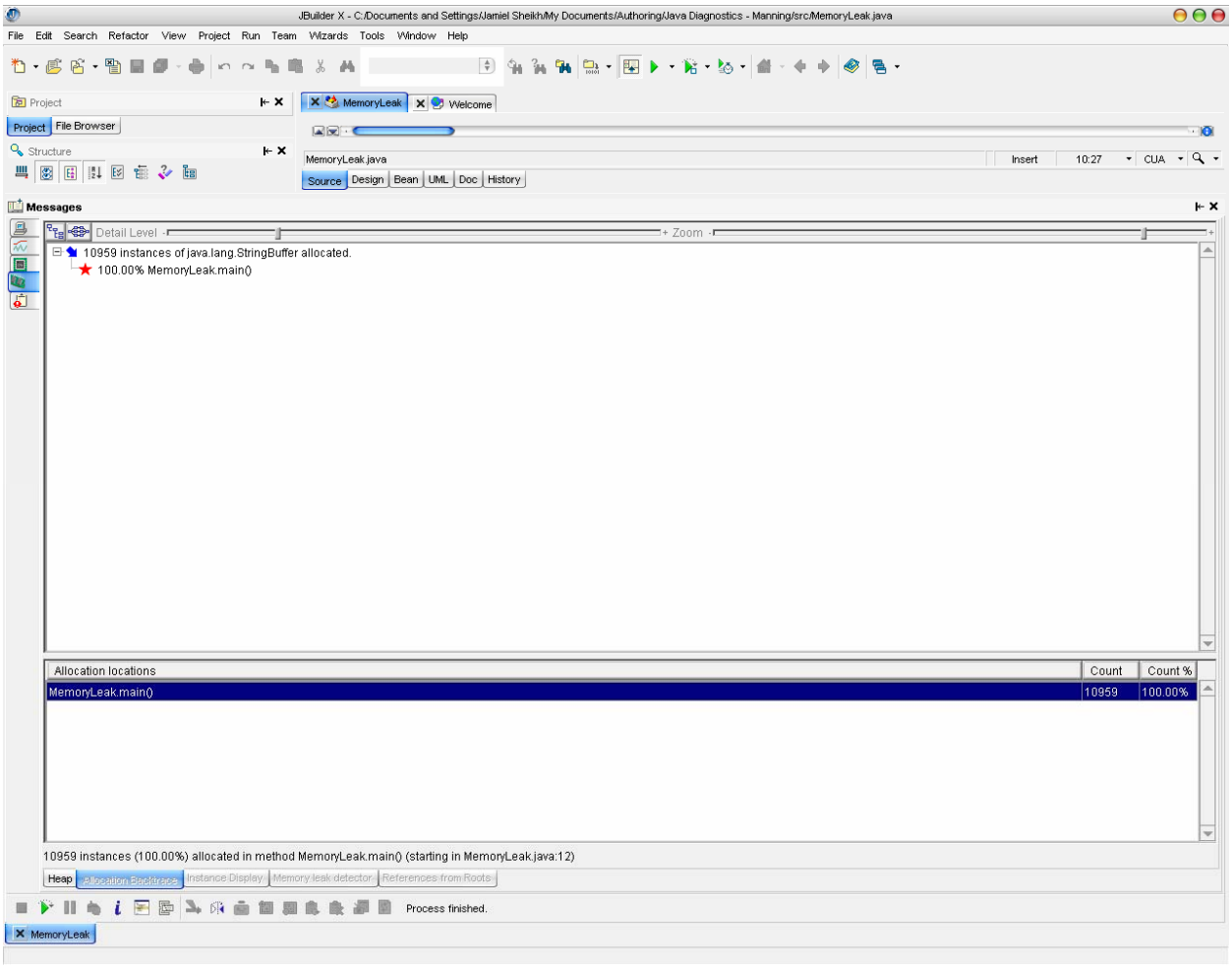

Figure 5.4 Optimizeit backtrace showing all `StringBuffer`s being allocated from the `main()` method of our class

After drilling down on the method, the profiler has completed its role. Your next course of action is to identify the actual offending code within that method. This means you must comb through each line of code within the given method(s) with some serious concentration. We recommend that a competent Java developer other than the one who authored the code be the reviewer; this allows for a fresh perspective when the code is reviewed. No matter how good a developer may be, mistakes are bound to be made. And more times than not, finding these mistakes is quicker and easier with a new pair of objective eyes.

After you successfully identify the bad code, you should make the appropriate changes and run the profiler again. Figure 5.5 shows another snapshot of heap graphs from Optimizeit. This time, however, you're viewing a healthy application, with the memory-leak removal lines uncommented in the original code. The upper-left frame shows that the maximum heap remains constant while the heap being used peaks and then returns to a baseline level. This cycle continues and is very much in synch with the activity of the garbage collector in a typical healthy application.

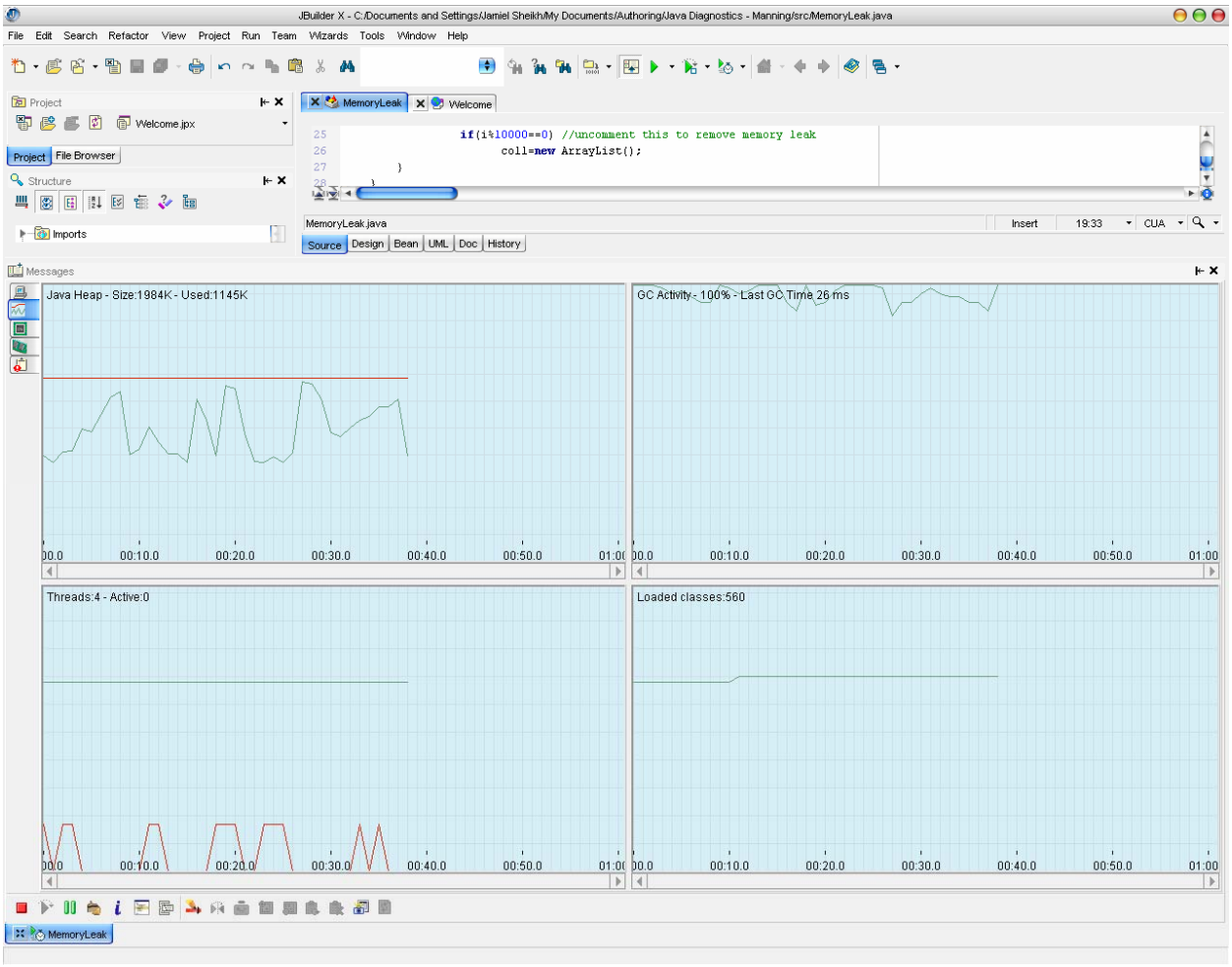

Figure 5.5 How a healthy application behaves after reaching a state of equilibrium. The usage of the heap grows but returns to equilibrium after garbage collection, and the cycle repeats itself.

As the example showed, it's a straightforward task to identify classes and objects that are the source of a memory leak. This task become more difficult as the scale of the memory leak is reduced, because the difference between the equilibrium object counts is smaller and requires more application execution time. Next, we'll show you how achieve the same goal using the freely available tool bundled with the Sun JVM, HPROF.