

Bitter memories

This chapter covers

- A Java memory-management primer
- Memory-related antipatterns
- Techniques for troubleshooting memory leaks

Kayakers and canoeists are not quite mortal enemies, but there's a healthy rivalry between them. As a child I canoed, but I made the switch to kayaks attracted by their stability, speed, and grace. I am now on an extended trip with three other kayakers--and one canoeist. The canoeist, Randy Barnes, and I spar verbally at every opportunity. I comment on his strong swimming skills, regretting that a canoe is so difficult to roll. Randy wonders aloud why I need the security blanket of a two-blade paddle when a canoeist can make do with a short, single blade. Our first day is spent on the Little River, a tight, technical run in the Smokey Mountains. I secretly marvel as Randy takes his canoe down a violent Class V rapid called the Sinks. I decide to walk around it; for me, the margin of safety is too slim. With outstanding control and incredible skill, Randy maneuvers his 12-foot canoe into places I have trouble taking my 7-foot kayak.

We are now running a Class IV+ rapid known as the Elbow. It's an extremely tight flume of water—in places just four feet wide. Bent tightly at two points, the Elbow drops nearly 20 feet down a 40-degree fall. Three kayakers at the bottom watch, stunned, as Randy's canoe hits the side of the chute, then the riverbed. The canoe tumbles all the way to the end of the run. Coughing up river water, Randy emerges as the kayakers jeer. Looking over the rapid as I prepare for my run, I remain silent.

6.1 **Understanding memory leaks and antipatterns**

In this chapter, we'll begin to explore antipatterns related to memory leaks. Some of you might think that memory-management discussions should be left to JVM vendors. Technically, a Java memory leak occurs when an object cannot be reached and it's not freed through an automatic process known as garbage collection. In this sense, the virtual machine vendors, rather than application programmers, should be the ones to concern themselves with memory leaks. However, we'll use the term *memory leak* in a much broader sense. For our purposes, an object that's not garbage-collected after it's no longer of use to an application is a memory leak. Some of you might consider this usage inaccurate, because Java is working precisely as it is designed. I chose the term memory leak because it best describes the behavior of the applications that suffer from these antipatterns: memory that we no longer need is not returned to the memory heap and will continue to “leak” away until we explicitly trace the source and fix the problem.

First, we'll examine memory management for other languages, such as C++. We'll also examine several garbage-collection techniques and identify the ones most JVMs use. In some places, we need to help the garbage collector to do its

job. To that end, we'll present a series of antipatterns involving memory leaks. Finally, we'll look at tools and techniques that will help us solve memory leaks.

6.1.1 Managing memory

In any object-oriented language, each variable, regardless of scope, must be allocated. In most cases, memory can come from three primary sources: registers, the stack, and the heap. The compiler (or interpreter) closely manages registers and the stack, so we'll focus on the heap. We can think of the heap as a mailbox with different-sized slots. A compiler or interpreter will have a memory manager, analogous to a postmaster, to manage these slots. Implementations of memory managers can vary widely, from single runtime entities to static libraries. It's the manager's job to configure the size of the slots, allocate them as needed, and reclaim them for later use. For some low-level languages, like C++, the manager is a very thin layer. The programmer is responsible for explicitly requesting a block of memory (with `new` or `malloc`) and explicitly freeing the block (with `free`) when it's no longer required. C++ also forces the programmer to track individual addresses of memory blocks through *pointers*. This implementation makes C++ very flexible, but also tedious and problematic. Table 6.1 shows some of the bugs that a C++ programmer might encounter. We'll focus on the dangling pointer and the memory leak, because both are realized differently in Java.

Table 6.1 Problems encountered while managing memory in C++. The third column shows the types of program failure that you can expect.

Name	Description	Failure Symptoms
Memory leaks	Memory is allocated. Memory is not freed.	This is the classic memory leak. The program grows indefinitely, slows as free memory runs out, and eventually fails with an allocation error.
Free errors	Memory is freed with no allocation.	The program fails with a free error.
Dangling pointers	Memory is freed, but the pointers not updated.	As lists, trees, or other structures are traversed, the program crashes with unpredictable results.
Invalid pointer; lost pointer	Pointers to valid memory addresses are overwritten.	The program crashes.

Recall that Java implements a different strategy. Java applications request objects, and the interpreter allocates the memory for the object. Because Java handles allocation for the programmer and all variables are represented as higher level objects, the Java programmer doesn't need to bother with pointer arithmetic. In fact, pointer arithmetic was explicitly avoided in the Java design to increase the stability and security of the language. Memory is not explicitly freed. Instead, Java handles deallocation of memory through a process called garbage collection.

6.1.2 **Understanding garbage collection**

Heaps are usually managed as a linked list or with a reference table pointing to blocks of memory. An allocated block is marked in the table or taken out of the chain. With allocation, a suitable block is found, subdivided, marked, and returned to the requesting application. To prevent fragmenting, some memory managers periodically coalesce memory or combine contiguous blocks into one big block. Automatic allocation of memory isn't difficult. The size of the memory block and the block usage are known in advance. The memory manager simply goes to the heap and walks through the free blocks until a block of sufficient size is found.

Garbage collection is the act of periodically finding unused memory and returning it to the heap. Doing it well is much more difficult than allocation. The garbage collector cannot reliably guess the programmer's intent, so it must use some other means to identify memory blocks to free. We'll discuss two important garbage-detection techniques: reference counting and unreachable nodes.

6.1.3 **Reference counting**

Early garbage collectors used a technique called *reference counting*. With this algorithm, each object has a counter. When an object is allocated or used, the associated counter is incremented. When the object is no longer referenced, it is freed. The garbage collector can periodically sweep through all allocated objects and free any objects with a reference counter of 0.

In figure 6.1, the circles represent allocated objects in the form of a *directed graph*, or a group of interconnected objects. Connections between objects are called *edges*. In our case, the edges are directed because an object that explicitly references another provides direction for our arrows. Our graph's objects are labeled with a name and reference count in parentheses. The objects above the line are in use, and the objects below the line are no longer in use.

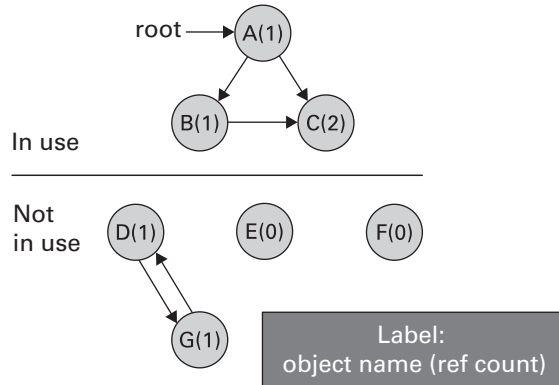


Figure 6.1 Some garbage collectors manage memory by counting references. Here, the objects with a reference count of 0 will be identified by the garbage collector and freed. Notice that objects D and G are not in use but will not be freed, due to a circular reference.

The problem with reference-counting garbage collection is that it cannot detect *cycles*, or objects that reference each other, though they might not be in use by any other object. In fact, we couldn't use the objects below the line if we wanted to, because we no longer have a reference. In our example, objects E and F would be freed, but D and G would not. D and G have positive reference counts, though we cannot possibly use them. This short program demonstrates an object that would be freed, and one that would not:

Listing 6.1 Demonstrating a circular reference

```

public class LinkedList
{
    public LinkedList next = null;
}
.
.
public someMethod () {
    LinkedList a = new LinkedList();
    LinkedList b = new LinkedList();
    a.link = b;
    b.link = a;
    a = null;
    b = null;
}
.
.
    
```

● **Linked list class**

● **Allocate A/B. A/B reference counters = 1.**

● **A/B reference B/A. A/B reference counters = 2.**

● **Remove references to A/B. A/B reference counters = 1.**



This program does nothing, but it illustrates the anatomy of a memory leak for programs that use reference counting for garbage collection. Many programmers believe that circular references can lead to memory leaks in Java, but Java garbage collectors no longer use reference counting. (Some C++ frameworks do still use reference counting for garbage collection, which might account for some of the confusion on the subject.)

6.1.4 Reachable objects

Java garbage collectors use another technique, called *reachable objects*, to determine whether an object is in use. The garbage collector periodically travels the directed graph of allocated objects, attempting to visit every node by following valid references. If an object can be reached, it is marked, and the rest are freed. With this two-pass approach, sometimes called a *mark-and-sweep* algorithm, circular references are handled appropriately: They are freed only if they aren't reachable, effectively handling the problems shown in figure 6.1.

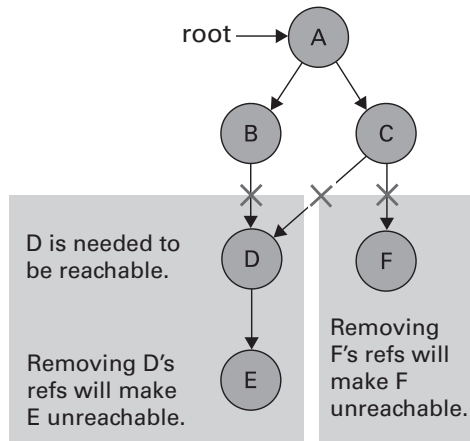


Figure 6.2 Java performs garbage collection by determining whether a node is reachable. An object cannot be garbage-collected if it is reachable. You can break reachability (D, F) by dereferencing an object or an object in the reference chain until it is no longer reachable (E).

6.2 Trading C++ for Java

Running through the Elbow after Randy, I'm glad to be in my tiny kayak. I am not expecting a problem making the S-turn, nor do I expect to hang up on the bottom. The two-blade paddle will help me brace on either side for stability. I need only make the initial critical turn, get into the flume, and brace. From that point on the turbulent water will steer me down the flume as it has the three kayakers before me. As I reach the initial turn, my front end hits turbulence—as I expected—but then skips over the water that I'd counted on to turn my boat. Out of control, I hit the same rocky outcropping as Randy. The mistakes are different, but the result is the same.

When I decided to eschew C++ and learn Java, I'd been frustrated with the amount of time required to handle tedious matters unrelated to my business problem. As a team leader, I'd spent hours tracking down dangling pointers, memory leaks, and off-by-one errors. C++ felt like a long, awkward canoe.

When I moved to Java, I felt *invulnerable* in my new “kayak.” I marveled in condescending glee at the things that Java handled for free, while my C++ friends struggled with memory-management issues. Then, I encountered my first Java memory leak, and I discovered that my tools and skills were not adequate. I hadn't known that Java memory leaks could even exist, let alone how to find them. It took me a week to solve my first Java memory leak. My memory-management days weren't behind me, after all.

6.2.1 Circumstances that cause Java memory leaks

Though garbage collection in Java is sophisticated, certain patterns can still cause memory leaks. An object's transition through states from allocated to freed is known as its *life cycle*. From a garbage-collection standpoint, an object goes from being unallocated, to allocated, to live, to unused, to garbage-collected. A live object is one that is being actively used by a program. Java memory leaks tend to occur when a reachable object with a long life cycle has a reference to an object with a shorter life cycle. In short, memory leaks are objects that are reachable, but not live. Figure 6.2 shows two ways that garbage collection can occur:

- All references to an object by all reachable objects must be removed. In figure 6.2, F has references from object C, which must be removed for F to be collected.
- All references to another object, needed for an object to be reachable, must be removed. In figure 6.2, references from B and C to D will

make object E unreachable, and will be collected on the next garbage-collection pass.

Eventually, chains are garbage-collected as they fall out of use. However, if object B or C has a long life cycle, then objects D and E will remain reachable unless the references to D are explicitly broken. If the programmer no longer intends to use D and E, we have a classic Java memory leak. Most of the memory leaks in this chapter will deal with some type of anchor with a very long life cycle that references unused objects, such as collections, singletons, and listeners. If nothing is done to remove the reference, we'll have a memory leak.

6.2.2 Finding Java leaks

Java leaks can be especially difficult to find. For the most part, Java applications are at a higher level than C++ applications. We Java programmers are not well versed with the low-level tools and techniques that complex memory problems demand. Factors that contribute to the dilemma of finding and troubleshooting memory leaks in Java include:

- *Simply watching memory by using a monitor, like the Windows Task Manager, is not sufficient.* Garbage-collection scheduling varies, and memory is not freed until garbage collection is scheduled.
- *Under certain circumstances, the Java garbage collector is not executed at all.* Some garbage collectors are activated only when memory use reaches a certain threshold. In those instances, if tests do not model real-world usage scenarios closely, you could easily miss memory leaks or incorrectly assume leaks exist.
- *References are in clusters, so the impact of leaks is greater.* In Java, each block of memory is not explicitly freed. If you forget to remove a critical reference to a single object, it may reference many other objects. A Java interface developed with the Swing component library can have references to dozens or even hundreds of child classes. Further, since the classes also have back pointers, references to a child class can make an entire user interface tree reachable.
- *More than one reference might be causing the leak.* Because all references must be broken to make a class unreachable and eligible for garbage collection, breaking a single reference might not be enough. A correct fix may not have the desired impact if multiple bugs exist.

- *The symptoms are not as dramatic or immediate as those in C++.* In C++, a reference is not correctly updated, the result is often a crash due to a dangling pointer. In Java, the symptoms are not immediate, so generating failure test cases is not as easy.

This chapter will help you recognize some common types of memory leaks before they become problems. Coding conventions make it easier for you to account for key references that must be explicitly removed to achieve effective memory management. Now that we've reviewed memory management and seen some basic patterns that could lead to Java memory leaks, let's examine some specific antipatterns.

6.3 **Antipattern: Lapsed Listeners Leak**

The Lapsed Listener Leak is a common antipattern given its name by Ethan Henry and Ed Lycklama in an article titled “How Do You Plug Java Memory Leaks?” One design pattern with the potential for memory leaks is the Event Listener design pattern, which is used to establish interest in an event without forcing broadcast messaging. Objects request notification of an event by registering a listener or method that will be called if the event occurs. Java uses this design pattern in several places; three are interesting for the purposes of this antipattern. First, the Java user interface framework (called the AWT library) uses this design pattern for notification of actions. Second, generic JavaBeans can use an interface called `PropertyChangeListener` to establish notification for a changing property. Third, the Java Swing component library uses the Model-View-Controller design pattern. The model is registered with a user interface component, and the user interface is notified when the model changes.

Three factors make event-notification design patterns ripe for memory leaks:

- The event registry can be a collection with a long life cycle. A user interface component that is registered cannot be garbage collected until it is removed or the root registry object (or *anchor*) is garbage collected. If the anchor has a long life cycle, garbage collection may not occur at all.
- The symptoms for failing to remove the notification are initially benign. The benign symptoms make it easy for the leak to get through initial tests.
- Many visual programming frameworks, Swing programming samples, and wizards do not remove listeners. Whether these samples are created with a development environment or through cut and paste, memory leaks are not expected from these sources.

6.3.1 Examining some dangerous practices

The program shown in listing 6.2 was created with VisualAge for Java. It's a simple applet called TestView. Notice that the development environment registered a method to handle the button action without a corresponding remove method.

Listing 6.2 Dangerously registering an action without a corresponding remove

```
package memory;

public class TestView extends java.applet.Applet {

class IvjEventHandler implements java.awt.event.ActionListener {
    public void actionPerformed(java.awt.event.ActionEvent e) {
        if (e.getSource() == TestView.this.getTest())
            connEtoM1(e);
    };
};
IvjEventHandler ivjEventHandler = new IvjEventHandler();
private java.awt.Button ivjTest = null;
private java.awt.TextField ivjTextField = null;
public TestView() {
    super();
}

private void connEtoM1(java.awt.event.ActionEvent arg1) {
    try {
        getTextField().setText("After");
    } catch (java.lang.Throwable ivjExc) {
        handleException(ivjExc);
    }
}

private java.awt.Button getTest() {
    if (ivjTest == null) {
        try {
            ivjTest = new java.awt.Button();
            ivjTest.setName("Test");
            ivjTest.setBounds(117, 166, 56, 23);
            ivjTest.setLabel("Test");
        } catch (java.lang.Throwable ivjExc) {
            handleException(ivjExc);
        }
    }
    return ivjTest;
}

private java.awt.TextField getTextField() {
    if (ivjTextField == null) {
        try {
```

● Trigger for an event

● Event handler class

● Method is fired when the event handler gets called.

```
        ivjTextField = new java.awt.TextField();
        ivjTextField.setName("TextField");
        ivjTextField.setText("Before");
        ivjTextField.setBounds(118, 100, 60, 29);
    } catch (java.lang.Throwable ivjExc) {
        handleException(ivjExc);
    }
}
return ivjTextField;
}

private void handleException(java.lang.Throwable exception) {
    exception.printStackTrace(System.out);
}

public void init() {
    try {
        super.init();
        setName("TestView");
        setLayout(null);
        setSize(426, 240);
        add(getTest(), getTest().getName());
        add(getTextField(), getTextField().getName());
        initConnections();
    } catch (java.lang.Throwable ivjExc) {
        handleException(ivjExc);
    }
}

private void initConnections() throws Exception {
    getTest().addActionListener(ivjEventHandler);
}
}
```

● **Event is registered here. Danger!**

In this case, VisualAge for Java has created event registrations with no corresponding remove methods. The application does not have a memory leak, because our registration class has a limited life cycle. However, we do have the foundation for one. If this code is reused with cut and paste, or if the life cycle of our registry changes, we'll have a leak.

How might the life cycle change? Suppose an application has a long-lived main window with many transient subwindows that are created and destroyed. Events registered to the class will prevent garbage collection of the child windows.

Another common implementation for object-oriented classes that can be shared is the singleton. With this design pattern, a single, shared object is created, typically with a long life cycle, and is used by many objects. Objects that

need to use the singleton get an instance. Consider the program shown in listing 6.3, which was taken from “Plugging Memory Leaks,” by Tony K.T. Leung. (Some cosmetic changes have been made for easier annotation.)

Listing 6.3 Leaking memory through registering to a singleton

```
import java.beans.*;

public class Test
{
    public static void main(String[] args)
    {
        C c = new C();
        c = null;
        System.gc();
    }
}

class C implements PropertyChangeListener
{
    private D d_ = null;

    public C ()
    {
        d_ = D.getInstance();
        d_.addPropertyChangeListener(this);
    }

    public void propertyChange(PropertyChangeEvent evt){}
}

class D
{
    private static D singleton_ = null;
    private PropertyChangeSupport listeners_ =
        new PropertyChangeSupport(this);
    private D(){}

    public static D getInstance()
    {
        if (singleton_ == null)
            singleton_ = new D();
        return singleton_;
    }

    public void addPropertyChangeListener(
        PropertyChangeListener listener)
    {
        listeners_.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener(
```

● **References C, removes reference, and then runs gc, suggesting garbage collection.**

● **A singleton, with a different life cycle from C.**

● **This reference will prevent garbage collection.**

● **Static instance variable; getInstance method. Singleton!**

● **Method registers the event (but there is no remove). Singleton!**

```
PropertyChangeListener listener)
{
    listeners_.removePropertyChangeListener(listener);
}
}
```

In this case, class C is registering a `propertyChanged` method with a static listener. Since D is a singleton (and has a long life cycle), it will remain reachable. Though the reference to C is removed in main, we have a memory leak. We can solve this solution in one of three ways.

6.3.2 **Solution 1: Explicitly remove the listeners**

You can solve this problem by explicitly removing listeners whenever you add them. With the graphical components, the places for removing listeners are well defined. The `Frame` and `Dialog` classes in the Java AWT library fire the `dispose()` method when the associated window is destroyed. Classes that add listeners and inherit from `Frame` or `Dialog` will also want to override the `dispose()` method and add a call to remove the event listener. For subclasses of components, cleanup can occur when the component is removed from the parent's container. This action fires the `removeNotify` method, which can be subclassed for the addition of the proper cleanup code. For property change listeners, the call is `removePropertyChangeListener`; for event listeners, the call is `removeActionListener`.

For other classes like the one in listing 6.3, there's no logical place. Since `finalize` is triggered by garbage collection and we're trying to remove references that will inhibit garbage collection, we have to invent a method and call it when we've finished using the class. It's better to place this code in close proximity to the code that registers the event listener. Both the add and the associated remove methods should be commented.

Finally, it's important to periodically verify that `addSomeKindOfListener` calls are paired with the associated `removeSomeKindOfListener` calls. If both lines are in close proximity, verification is simple. If not, the calls can be paired with a text search, like `grep`.

6.3.3 **Solution 2: Shorten the life cycle of the anchor**

One way to ensure that a listener will be garbage collected is to make sure that the listener registry can be garbage collected. Listing 6.2 does not have a memory leak because the registry is a short-lived component. This is usually not the best solution by itself, because it leaves `addActionListener` calls

without the matching `removeActionListener` calls, which means the program is vulnerable to future memory leaks. It also takes a singleton class and changes it to a regular object, which must be instantiated for each use. The singleton may have been initially created to conserve system resources.

6.3.4 **Solution 3: Weaken the reference**

With the introduction of version 1.2, Java offers additional types of references that can be used to solve this problem. In addition to the standard object reference, Java has specialized *reference objects* that can work with the garbage collector in special circumstances. There are three types: weak, soft, and phantom references. For this solution, we need weak references. To the garbage collector, an object that can be reached only by using a weak reference is called *weakly reachable* and is a candidate for garbage collection. In figure 6.3, objects C and D are weakly reachable. Weak references can be placed in collections with longer life cycles, in situations that might normally inhibit garbage collection. Because objects can be garbage collected, the null condition should always be checked whenever you're accessing an object through a weak reference.

6.3.5 **Reference objects simplify memory management**

Each solution has its advantages and disadvantages. Weak references provide a generic solution that's suitable for frameworks and tools, like visual programming environments, but it isn't as clean a solution as explicitly removing each

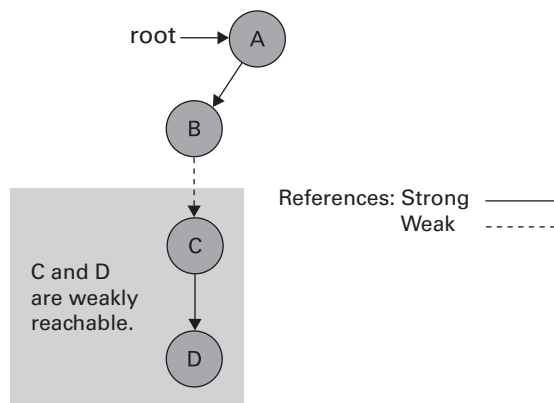


Figure 6.3 Weak references can be used to assist the garbage collector. An object is weakly reachable if it can be reached only through a weak reference. Weakly reachable objects are candidates for garbage collection.

listener that's added. On the other hand, weak references do add to our bag of tricks for memory management. Table 6.2 shows the strengths and weaknesses of each solution.

Table 6.2 These are the solutions to the Lapsed Listener antipattern. Each circumstance is unique, and each solution has pros and cons that must be considered.

Solution	Strengths	Weaknesses
Balancing removes and adds	The design is clean, readable, and intuitive. Implementing graphical components is straightforward.	The placement of the remove is not always intuitive, especially for objects that are not components.
Shorten the anchor's life cycle	The solution can be very easy to implement and is easy to tool.	The solution is vulnerable to bugs, resulting in memory leaks.
Weak references	This is the most generic solution, and is also very easy to tool.	The code for weak references is not as intuitive and readable as direct references.

6.4 Antipattern: The Leak Collection

The root of the Lapsed Listener antipattern is a collection with a long life cycle that has object references inserted but not removed. The references prohibit effective garbage collection because they make the referenced object reachable. The Leak Collection is a general case of the Lapsed Listener antipattern. Whenever we have a collection with a long life cycle that has the potential to contain objects that aren't removed, we have the possibility of a memory leak. Figure 6.4 shows the basic antipattern.

In essence, an application (A) allocates an object (B) with a short life cycle, registers the object in a collection (C) having a relatively long life cycle, and then removes the reference to object B. If the collection has a long life cycle—as many shared collections do—and if the object isn't removed from the collection before the reference is garbage collected, we'll have a memory leak.

Many different design patterns and problem domains call for singleton collections. Here are a few:

- Caches: A singleton hash table is the implementation of choice for a cache. When a cache doesn't have a policy to expire old entries, we have the potential for a leak.

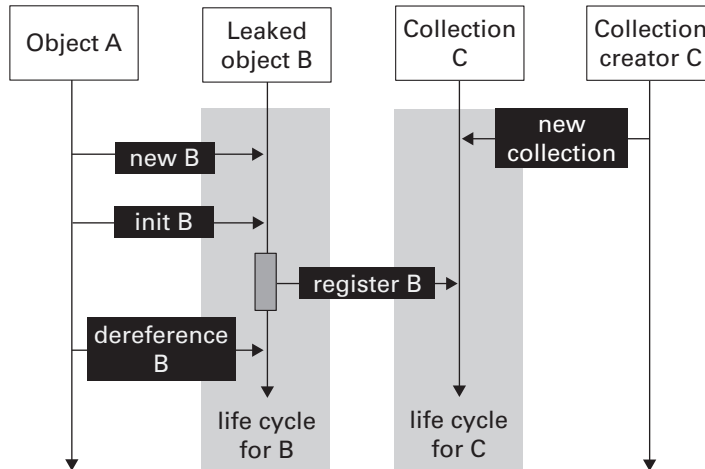


Figure 6.4 In the Leak Collection antipattern, an object with a short life cycle is registered in a collection with a long life cycle. Here we've shown a UML sequence diagram of such a memory leak. In this case, object B will be a memory leak, because the reference in collection C won't allow the garbage collector to free it.

- Session state: A well-publicized Tomcat bug turned out not to be a bug at all, but a session-management problem. By default, session management was turned on, and many didn't know it. In certain circumstances, the session dictionary grew until memory was gone. In general, session state is a good place to find a memory leak.
- User interfaces: Most modern user interfaces are collections of windows and components. Many times, these potentially massive collections have persistent anchors. Some common places to find singleton anchors might be print support, font management, and interapplication communication. Whenever a user interface is attached to an anchor with a long life cycle, leak conditions are present.
- EJB containers: An EJB container is essentially a smart singleton collection of EJBs.

6.4.1 **Causing trouble with caches and session state**

Two specialized uses of collection classes are especially troublesome. Session state captures the conversation context between the client and the server. Because HTTP is a *stateless protocol*, the infrastructure or the application must manage conversation details. The duration of Internet conversations varies wildly among consecutive communications from a given client, so the

management of session state is particularly difficult. Currently, standard specifications for the various servlet protocols deal with expiration of session state through timeout. When a session is established, a timeout is specified. When the timeout expires, the conversation expires as well. Excessive timeout lengths can cause memory to be exhausted, while too short a timeout can cause frustrating user experiences.

A similar problem involves the cache. In many cases, a cache can exhaust memory if the data set is large and older data is not removed periodically. If the cache is allowed to grow unabated, memory will eventually be exhausted. With only minor changes, we can easily ensure that cache memory can be neatly managed.

6.4.2 **Solution 1: Search for common warning signs**

Collections can serve many different purposes and can be involved in many different types of leaks, but you can recognize common threads. Table 6.3 describes the common warning signs, along with the appropriate actions.

Table 6.3 Memory leaks are common in applications with certain characteristics.

Warning Sign	Description	Action
Mismatched life cycles	Whenever an object with a long life cycle references an object with a short life cycle, there is leak potential.	Examine objects with long life cycles. Try to make sure references to transient objects are removed.
Mismatched add/remove for shared collections	Whenever a shared collection exists, adds without deletes may provide references that prohibit garbage collections.	Make sure that any add has a corresponding delete or weak reference that allows garbage collection.
Singletons and static objects	Static objects and singletons have long life cycles, and have the potential for memory leaks.	Examine static objects and singletons to make sure that references to transient objects are removed and commented.
“May” conditions for registrations	Whenever a registration to a collection is voluntary (“may” versus “must”), absence of strict attention can lead to a memory leak.	Watch voluntary registrations especially closely, or weaken the references to allow garbage collection.
APIs that hide collections	Whenever an API hides an addition to a collection, it is possible that the associated remove will be missed.	Comment APIs that hide collections, and make sure that responsibilities are clear, or weaken the references to allow garbage collection.

The main indications that conditions are common for a leak are mismatched add/remove pairs, mismatched life cycles, and other kinds of anchors with longer life cycles. Singletons and static classes can often provide safe harbor for memory leaks.

6.4.3 **Solution 2: Aggressively pair adds with removes**

To effectively deal with collection-based memory leaks, you might have to change your programming hygiene. Consider collection management with all CRUD operators: Create, Read, Update, and Delete. Any time you add a row to a collection, you need to add the corresponding remove. In fact, you should do both at the *same time* so you don't forget. Further, if possible, ensure that the pairs are in close proximity in the code. In addition, you can tag the pairs with a comment or with the name of the collection so that you can easily use strategies to look for pairings when you have to refactor or deal with a memory leak. Then, you can use tools such as `grep` to scan the program and look for pairs. In some cases, proximity may not be an option, like the use of a cache. For these collections, you can use a different utility to periodically prune the cache to remove elements that have timed out. In these cases, different strategies will be required.

6.4.4 **Solution 3: Use soft references for caches**

Though inconsistent implementation diminishes their value, caches (and session state managers, a special case of the cache) are collections that demand a different approach. We'd like the cache to use all available memory, and be freed only when the system needs the additional resources. Fortunately, Java soft references provide a mechanism that can do exactly that. An object is said to be *softly reachable* if it can be reached only through use of a soft reference. The Java 1.3 specification says that softly reachable objects can be freed at the garbage collector's discretion with two caveats:

- The garbage collector will attempt to free soft references before throwing an out-of-memory exception.
- The garbage collector should attempt to free soft references in least recently used order.

These properties make soft references ideal for the implementation of caches. In practice, some garbage collectors treat soft references like weak references, diminishing their value in caches. Clearly the intent of the soft reference is for use in such applications as memory-sensitive caches. While the JVMs should attempt to free soft references in the least recently used order, many do not.

6.4.5 **Solution 4: Use collections with weak references**

The Java 1.2+ specifications build in some collections that implement weaker references. `WeakHashMap` is a class that implements a hash table with weak references. Objects referenced solely from this hash table are weakly reachable. This type of collection is ideal for the optional registration of objects, and for applications where it's difficult to pair add/remove method calls managing the items in the hash table.

6.4.6 **Solution 5: Use finally**

Even when the calls are placed appropriately, exceptions can prevent the proper cleanup from ever occurring. Important cleanup should be placed in `finally` blocks. The Java `finally` block is executed after all code in a method, including exception management, is processed. `finally` will guarantee that a necessary block of code is executed, and is especially useful for cleanup.

6.5 **Shooting memory leaks**

These antipatterns can be used to identify existing leaks through inspection, but what techniques can help you troubleshoot applications with existing memory leaks? C++ programmers use spectacularly cumbersome tools and techniques to solve memory-related problems. For the most part, Java programmers prefer to stay above the fray. Sometimes, however, we aren't so fortunate. In this section, we'll look at the strategies that can help. The basic steps to finding memory leaks are determining whether a problem exists, isolating the problem, repairing the problem, and protecting against the problem in the future.

6.5.1 **Make sure there is a leak**

Much is left to the discretion of garbage collectors. In some cases, garbage collection can occur relatively frequently. In other cases, it may not run at all. Many of the faster JVMs try to limit the number of times that garbage collection is run. Some JVMs execute garbage collection only when memory resources are close to exhausted. In such cases, an application can appear to leak. Watching available memory on a simple memory monitor is not enough. A good strategy is to take memory snapshots before and after suggesting garbage collection with a call to `System.gc`. (This method call should seldom or never be used in production code.) Many tools can force garbage collection.

Tools are available to help you analyze memory leaks. A few are J-Insight (from alphaWorks), J-Probe (from Sitraka Software), and OptimizeIt (from

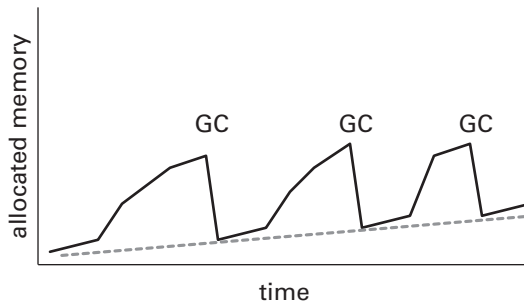


Figure 6.5 This is the classic resource chart of a memory leak. The solid black line shows actual memory use. The broken line projects reachable memory objects. The peaks and valleys are caused by garbage collection. The slowly rising valleys are characteristic of memory leaks.

Intuitive Systems). A well-behaved application should show a usage pattern that builds to peaks and then has distinct valleys. With similar types of activities, the valley floors should be consistent. The peaks represent normal object allocation over time. The valleys represent memory use just after garbage collection. If the valley floor continually creeps up between garbage-collection cycles, that's a sign of a possible memory leak. Figure 6.5 shows the graph of a classic memory leak. The GCs denote garbage collection at each of the peaks. The dotted line shows a slowly increasing amount of memory not garbage collected.

Sometimes, the symptoms of a production memory leak are clear. Of course, a `java.lang.OutOfMemoryError` is a clear sign of a leak, but the exception is not always generated before a more catastrophic system error occurs. Other symptoms may also point to a memory leak. In many cases, a leak will cause a relatively smooth decline with progressively worsening performance. As the system begins to swap or page more, the system performance will degenerate rapidly until the application fails or the system crashes. Low memory can cause many different types of failures, such as an inability to connect to a database or open a file, even though the root cause is lack of memory.

6.5.2 Determine that the leak should be fixed

Fixing memory leaks can be expensive. In some cases, we should push leak fixes lower on the priority chain, and maybe not even bother to do them. If an application's in-memory life span is relatively short, it may not make sense to patch all memory leaks, because garbage collection may never be run. If memory leaks and total memory usage is small, memory leaks may not be a

concern. Table 6.4 offers some guidance as to whether memory leaks should be fixed at all.

Table 6.4 In some circumstances, fixing a memory leak may not be worth the effort. This table shows common memory leak characteristics, and provides a guideline for whether the fix will be worthwhile.

Application Characteristic	Fix Memory Leak?	Comments
Short application life cycle.	Seldom	Garbage collection may never be run.
Very small footprint.	Seldom	The leak should be fixed only if the application has a long enough life for the cumulative leaks to be a problem.
Fixing hurts code readability.	Sometimes	Readability is not the overriding concern, but may discourage fixing a leak in some circumstances.
Footprint is large.	Often	The leak should be fixed if the application life cycle is long or if resources are likely to become a problem.
Fix is very easy.	Yes	A memory leak is a bug. If the fix has been identified and has no other side effects, it should be fixed.
Code runs 24x7.	Yes	The leak will eventually become a problem.
Code is a target for reuse.	Yes	Regardless of other characteristics, if the code is a reuse target, it should be fixed

You should be especially vigilant about fixing leaks in two instances: when an application runs 24x7, and when code is a target for reuse. Leaks in either circumstance should nearly always be plugged. In any case, to prevent the spread of bugs through cut and paste, if a leak exists and you don't fix it, then you should at least document it.

6.5.3 *Isolate the problem*

Once you've determined there is a leak, and that the leak should be fixed, you need to isolate a test case that creates the problem. Effective testing can be much more productive than the use of low-level memory debuggers, so it's important to make the test case as narrow as possible. Unfortunately, that isn't always possible. The complexity of user scenarios and the degree of coupling between elements of the application will collectively determine how effectively

a memory leak can be isolated through scenarios. After you've isolated a leak, add the test to the application's regression test suite.

6.5.4 **Determine the source and fix the problem**

Once you've generated the narrowest possible test case, the next step is to determine the source of the problem. Without a doubt, this is the most difficult part of the process. These strategies may help:

Use good tools

You must have effective tools to be able to track memory leaks in Java. You need tools that let you:

- Trigger garbage collection on demand.
- Examine the size of the heap collectively over time.
- Examine the contents of the heap, including objects on the heap.
- Determine the references to an object (that prevent garbage collection).

Inspect code by hand

Many times, you can locate memory leaks by looking in the likely places. With close attention to collections, listeners, singletons, and long life cycles in general, you can spot many possible leaks. Further, searching tools can aid manual inspection. Simply counting the number of adds and removes to a problematic collection using `grep` or an editor can help you identify an imbalance that may lead to a leak.

Force garbage collection between repeated test cases

Most good profiling tools can force garbage collections. In this way, a code segment can be repeatedly exercised, garbage collection run, and the heap examined for growth. With the profiler's snapshot of the heap (graphs of allocated space versus free space over time work well), you can make the test case narrower.

Use object reference graphs when the search is sufficiently narrow

Most good profilers will show the references from an object on the heap. Since you're looking for reachable objects not garbage collected, profiling is a powerful tool when combined with garbage collection on demand. When you find a reachable object that should have been garbage collected, you can follow the reference chain to find the culprit.

Iterate

When you find a problem, fix it and start over. Patching one leak can solve others as well. Many leaks are interrelated—sometimes in surprising ways—and you’ll find it much easier to fix links one at a time than to try to solve them all with a single pass.

Fixing leaks usually turns out to be relatively easy after you’ve identified a problem. You must make the object unreachable by using one of the techniques described in this chapter. These techniques can make an object unreachable, and thus a candidate for garbage collection:

► **Fixing Java application memory leaks**

When an object with a long life cycle maintains a reference to an object with a short life cycle, we have the potential for a memory leak. These possible fixes remove references to make an object unreachable and eligible for garbage collection.

- Remove the object reference directly by setting it to another value.
- Remove an object from a collection.
- Weaken the references using Java reference objects.
- Shorten the life cycle of the referent.
- Shorten the life cycle of the object.
- Remove the object from the code.
- Refactor the code.

6.5.5 Protect against the problem for the future

When coding problems occur, you should examine the cause. If the problem is an isolated case of programming error, no further action may be necessary. These steps will help keep the problem from occurring in the future:

- Add to the test suite to make sure that the new case is covered. This will ensure that if the problem crops up again through poor change control or cut and paste, it will be found.
- If the problem has occurred in the past, an antipattern should be documented and shared.
- If the antipattern or solution lends new significant insight, it should be published at some level. Publishing paths can be as focused as emails to peers or presentations in department meetings, and as broad as writing a book or speaking at a conference.

Antipatterns work best when we establish a pattern, solve a problem in a general way, and disseminate the wisdom.

6.6 **Mini-Antipatterns: Little Hogs**

One little hog cannot eat much, but many little ones can have devastating appetites. This chapter title may have led you to expect a whole chapter on dealing with strings and the little things that a programmer can do to save a few bytes here and there. This section is the last in the chapter because I do not think that little optimizations should be emphasized, especially in cases where they can affect performance. There are some places where memory can become an issue, such as in pervasive environments, very large object trees, and massive collections of objects. In these extreme cases, it makes sense to consider the microtechniques, which can collectively yield considerable savings. In other cases, I do not advocate reckless or careless memory use. I simply prefer readability to a few bytes of memory savings, and trust that I can make up the difference by optimizing the most important test cases first. With that in mind, let's consider some of the little memory hogs that can add up. In most cases, performance problems will be found in higher level designs, so low-level optimizations should be saved for later.

6.6.1 **String manipulation**

In memory-constrained situations, you should pay close attention to the proliferation of strings. A common offender is the + operator. If we string several + operators together to build a string, then each string argument will allocate an additional object and force more than one copy operation. This can get expensive, in terms of memory and performance. To clarify, consider the two following string treatments:

```
String s = "this code uses plus to break between "+
           "lines but doesn't result in any extra "+
           "objects as it still counts as a compile-time"+
           "constant";
```

This string works. The optimizer will allocate one `StringBuffer`, build the string up there, and then turn it into a single string. The following is much more dangerous:

```
String x = "Hello ";
x += name;
x += ", your birthday is ";
x += birthday;
```

This fragment will take significantly longer to execute, because in this case, each individual string will be allocated and copied. It does have a memory impact, although it is negligible. The bigger impact is the cumulative time that

it takes to reallocate and copy each individual string. In Java, this antipattern frequently occurs within loops that build queries, process parameters, or build XML documents. Instead, you should use a single `StringBuffer` and append each successive string into the buffer. Alternatively, you can keep the string together:

```
String x = "Hello " +
    name +
    ", your birthday is " +
    birthday;
```

In this way, you let the optimizer do the work for you. It will allocate a single string buffer to process the entire string, saving memory and plenty of CPU cycles.

6.6.2 Collections

We have seen that collections can have a dramatic impact on memory, because they tend to be used to manage objects in large numbers. Collections can affect memory in other ways as well. For example, large multidimensional arrays can consume staggering resources. Collection choice can also affect memory and performance.

Collections and allocation

Different Java types handle memory resources differently. Arrays allocate memory when they're created. Other collection types, such as sets, vectors, hash tables, and lists, allocate memory when items are added. Many times, preallocation is a good thing. If you know that you'll be allocating an explicit number of objects, that you'll be accessing the collection randomly or exclusively by a numerical index, and you know precisely when the resource will be used, then an array could be a good choice. Other times, preallocation can cause you to make incorrect assumptions or allocate much more memory than you are likely to use. Beginners frequently prefer arrays to more robust collections. If you know that you'll be adding a variable number of objects throughout the life cycle of the collection, or that access will be random by some other key, then a hash table may be a better choice.

Collections and access patterns

The type of access will also have an impact on the collection choice. If you must frequently access by a key other than an index, then a hash table, dictionary, or b-tree may be a better data structure than an array. If the order does not matter, a set may be a better choice. The key for success is to pick an

abstraction that fits the collection type. If the size is fixed and you need to frequently enumerate the collection, an array could be the best choice.

Arrays

Because arrays can have a significant effect on memory, they warrant special consideration. These tips can help:

- If you have many collections with widely varied size, consider another collection that allocates dynamically.
- Delay initialization (and thus allocation) until the automatic variable is needed.
- If the array is sparse, it should be declared appropriately. You shouldn't create methods (such as accessors) that you don't plan to use.

6.6.3 Inheritance chains

We have all seen excruciatingly complex object hierarchies with inheritance chains that reach all the way to China. Similarly, just as databases can be normalized too far, object-oriented design can be taken to extremes beyond anything that can practically perform. If you aren't in a memory-constrained environment, let common sense be your guide. On the other hand, as inheritance chains get longer, the memory cost is higher. If memory is a serious concern and other avenues have been exhausted, reducing the length of inheritance chains can save some valuable memory. This practice should never compromise reuse, design principles, or readability, but when considering whether to add one more subclass, you'll find that memory can be a tiebreaker.

6.7 Summary

Let's clean up this chapter by reviewing what we have covered and presenting our antipattern templates. We began by discussing memory-management philosophies of C++ and Java. We presented the old and new strategies of garbage collection, and showed that Java garbage collection is based on a concept called reachable objects. We then showed that even Java is prone to memory leaks, and we described general symptoms and specific antipatterns called Lapsed Listeners, Leak Collections, and Little Hogs. Finally, we examined common strategies that you can use to shoot down memory leaks, and looked at some "little hogs" that can make a big cumulative impact.

6.8 **Antipatterns in this chapter**

These are the templates for the antipatterns that appear in this chapter. They provide an excellent summary format and form the basis of the cross-references in appendix A.

Lapsed Listeners

DESCRIPTION: The Publish/Subscribe design pattern requires applications or classes with an interest in an event to register. The Lapsed Listener is one form of memory leak where an event listener is registered without being removed. If the life cycle of the listener registry is long, then a memory leak will occur.

RELATED ANTIPATTERNS: The Leak Collection. The Lapsed Listener is a special case of the more general Leak Collection.

MOST FREQUENT SCALE: Application.

REFACTORED SOLUTION NAME: Weak References, or Pairing Register with Remove.

REFACTORED SOLUTION TYPE: Software.

REFACTORED SOLUTION DESCRIPTION: One solution to this problem is to explicitly remove the listener. For clarity, register and remove listeners in add/remove pairs. If this cannot be done in the same method, the two methods should be in close proximity. Another solution is to weaken the reference with Java weak reference objects.

TYPICAL CAUSES: Programming hygiene is a common cause. When registrations are not placed in close proximity to removes, it is easy to neglect the remove, because the symptoms are delayed.

ANECDOTAL EVIDENCE: “I didn’t know you could have a memory leak in Java.” “The system gets slower and slower, and then it hangs or traps.”

SYMPTOMS, CONSEQUENCES: Some objects are not garbage collected, even though their primary user is. This leak will cause the system to slow over its life cycle, until it’s terminated or eventually dies.

SOLUTION ALTERNATIVES: Remove the reference appropriately or shorten the life cycle of the registry. Adding a listener without a weak reference or a corresponding remove is possible if the life cycle of the registry is short, but this is vulnerable to changes in life cycle and cut and paste.

Leak Collections

DESCRIPTION: If a collection has a long life cycle, it can have long-lived references that are never removed. These will prevent large blocks of memory from being freed.

MOST FREQUENT SCALE: Application.

REFACTORED SOLUTION NAME: Weak References, or Pairing Add with Remove.

REFACTORED SOLUTION TYPE: Software.

REFACTORED SOLUTION DESCRIPTION: One solution to this problem is to explicitly remove the reference from the collection. For clarity, add and remove objects in pairs. If this cannot be done in the same method, the two methods should be in close proximity. Another solution is to weaken the reference with Java weak reference objects.

TYPICAL CAUSES: Programming hygiene is a common cause. When adds are not placed in close proximity to removes, it is easy to neglect to remove, because the symptoms are delayed.

ANECDOTAL EVIDENCE: “I didn’t know you could have a memory leak in Java.” “The system gets slower and slower, and then it hangs or traps.”

SYMPTOMS, CONSEQUENCES: Some objects are not garbage collected, even though their primary user is. This leak will cause the system to slow over its life cycle, until it’s terminated or eventually dies.

SOLUTION ALTERNATIVES: Remove the reference appropriately or shorten the life cycle of the registry. Adding a listener without a weak reference or a corresponding remove is possible if the life cycle of the registry is short, but this is vulnerable to changes in life cycle and cut and paste.