Java Reference Objects

or
How I Learned to Stop Worrying and Love OutOfMemoryError

Introduction

I started programming with Java in 1999, after fifteen years with C and C++. I thought myself fairly competent at C-style memory management, using coding practices such as pointer handoffs, and tools such as Purify. I couldn't remember the last time I had a memory leak. So it was some measure of disdain that I approached Java's automatic memory management … and quickly fell in love. I hadn't realized just how much mental effort was expended in memory management, until I didn't have to do it any more.

And then I met my first `OutOfMemoryError`. Just sitting there on the console, with no accompanying stack trace … because stack traces require memory! Debugging that error was tough, because the usual tools just weren't available, not even a `malloc` logger. And the state of Java debuggers in 1999 was, to say the least, primitive.

I can't remember what caused that first error, and I certainly didn't resolve it using reference objects. They didn't enter my toolbox until about a year later, when I was writing a server-side database cache and tried using soft references to limit the cache size. Turned out they weren't too useful there, either, for reasons that I'll discuss below. But once reference objects were in my toolbox, I found plenty of other uses for them, and gained a better understanding of the JVM as well.

The Java Heap and Object Life Cycle

For a C++ programmer new to Java, the relationship between stack and heap can be hard to grasp. In C++, objects may be created on the heap using the `new` operator, or on the stack using "automatic" allocation. The following is legal C++: it creates a new `Integer` object on the stack. A Java compiler, however, will reject it as a syntax error.

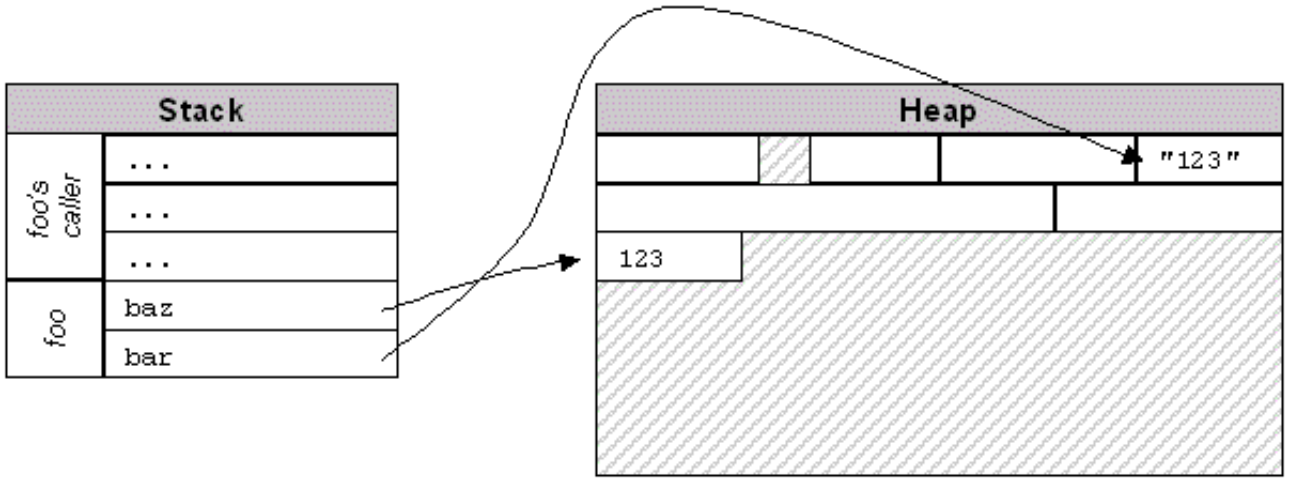
```
Integer foo = Integer(1);
```

```
Integer foo = Integer(1);
```

Java, unlike C++, stores all objects on the heap, and requires the `new` operator to create the object. Local variables are still stored on the stack, but they hold a pointer to the object, not the object itself (and of course, to confuse C++ programmers more, these pointers are called “references”). Consider the following Java method, which has an `Integer` variable that references a value parsed from a `String`:

```
public static void foo(String bar)
{
    Integer baz = new Integer(bar);
}
```

The diagram below shows the relationship between the heap and stack for this method. The stack is divided into “frames,” which contain the parameters and local variables for each method in the call tree. Those variables that point to objects — in this case, the parameter `bar` and the local variable `baz` — point at objects living in the heap.



Now look more closely at the first line of `foo()`, which allocates a new `Integer` object. Behind the scenes, the JVM first attempts to find enough heap space for this object — approximately 12 bytes on a 32-bit JVM. If able to allocate the space, it then calls the constructor, which parses the passed string and initializes the newly-allocated object. Finally, the JVM stores a pointer to that object in the variable `baz`.

That's the “happy path.” There are several not-so-happy paths, and the one that we care about is when the `new` operator can't find those 12 bytes for the object. In that case, before giving up and throwing an `OutOfMemoryError`, it invokes the garbage collector in an attempt to make room.

Garbage Collection

While Java gives you a `new` operator to allocate objects on the heap, it doesn't give you a corresponding `delete` operator to remove them. When method `foo()` returns, the variable `baz` goes out of scope but the object it pointed to still exists on the heap. If this were the end of the story, all programs would quickly run out of memory. Java, however, provides a garbage collector to clean up these objects once they're no longer referenced.

The garbage collector goes to work when the program tries to create a new object and there isn't enough space for it in the heap. The requesting thread is suspended while the collector looks through the heap, trying to find objects that are no longer actively used by the program, and reclaiming their space. If the collector is unable to free up enough space, and the JVM is unable to expand the heap, the `new` operator fails with an `OutOfMemoryError`. This is normally followed by your application shutting down.

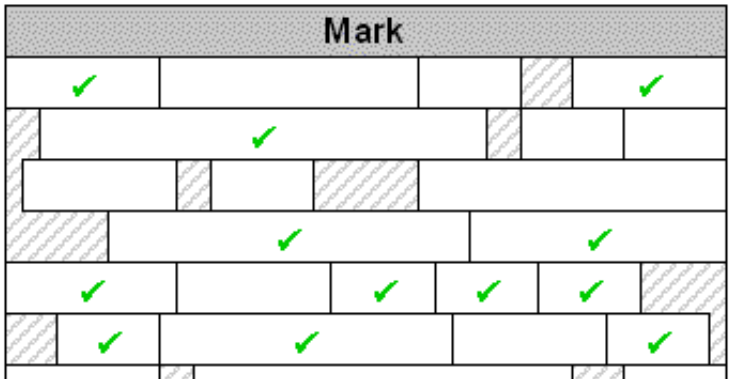
Mark-Sweep

One of the enduring myths of Java revolves around the garbage collector. Many people believe that the JVM keeps a reference count for each object, and the collector only looks at objects whose reference count is zero. In reality, the JVM uses a technique known as “mark-sweep.” The idea behind mark-sweep garbage collection is simple: every object that can't be reached by the program is garbage, and is eligible for collection.

Mark-sweep collection has the following phases:

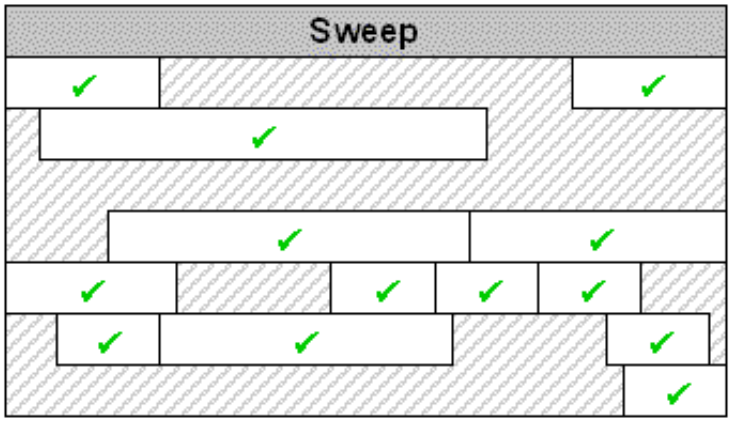
Phase 1: Mark

The garbage collector starts from “root” references, and walks through the object graph marking all objects that it reaches.



Phase 2: Sweep

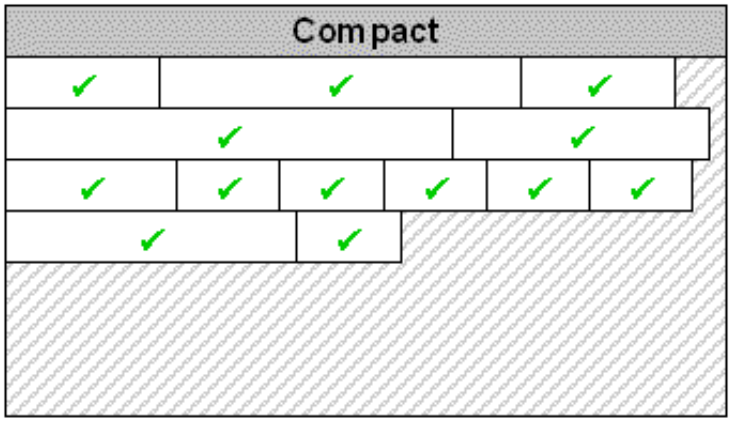
Anything that hasn't been marked in the first phase is unreachable, and therefore, garbage. If a garbage object has a finalizer defined, it's added to the finalization queue (more about that later). If not, its space is made available for re-allocation (exactly what that means depends on the specific GC implementation, and there are many implementations).



Phase 3: Compact (optional)

Some collectors have a third step: compaction. In this step, the GC moves objects to coalesce free space left behind by the collected garbage. This prevents the heap from becoming fragmented, which can cause large contiguous allocations to fail.

The Hotspot JVM, for example, uses a compacting collector for its “young” generation, but a non-compacting collector (at least in the 1.6 and 1.7 “server” JVMs) for its ”tenured” generations. For more information, see the references at the end of this article.



So what are the "roots"? In a simple Java application, they're method arguments and local variables (stored on the stack), the operands of the currently executing expression (also stored on the stack), and static class member variables.

In programs that use their own classloaders, such as app-servers, the picture gets muddy: only classes loaded by the system classloader (the loader used by the JVM when it starts) contain root references. Any classloaders that the application creates are themselves subject to collection, once there are no more references to them. This is what allows app-servers to hot-deploy: they create a separate classloader for each deployed application, and let go of the classloader reference when the application is undeployed or redeployed.

It's important to understand root references, because they define what a "strong" reference is: if you can follow a chain of references from a root to a particular object, then that object is "strongly" referenced. It will not be collected.

So, returning to method `foo()`, the parameter `bar` and local variable `baz` are strong references only while the method is executing. Once it finishes, they both go out of scope, and the objects they referenced are eligible for collection. Alternatively, `foo()` might return a reference to the `Integer` that it creates, meaning that object would remain strongly referenced by the method that called `foo()`.

Now consider the following:

```
LinkedList foo = new LinkedList();
foo.add(new Integer(123));
```

Variable `foo` is a root reference, which points to the `LinkedList` object. Inside the linked list are zero or more list elements, each of which points to its successor. When we call `add()`, we add a new list element, and that list element points to an `Integer` instance with the value 123. This is a chain of strong references, meaning that the `Integer` is not eligible for collection. As soon as `foo` goes out of scope, however, the `LinkedList` and everything in it are eligible for collection — provided, of course, that there are no other strong references to it.

You may be wondering what happens if you have a circular reference: object A contains a reference to object B, which contains a reference back to A. The answer is that a mark-sweep collector isn't fooled: if neither A nor B can be reached by a chain of strong references, then they're eligible for collection.

Finalizers

C++ allows objects to define a destructor method: when the object goes out of scope or is explicitly deleted, its destructor is called to clean up the resources it used. For most objects, this means explicitly releasing any memory that the object allocated with `new` or `malloc`. In Java, the garbage collector handles memory cleanup for you, so there's no need for an explicit destructor to do this.

However, memory isn't the only resource that might need to be cleaned up. Consider `FileOutputStream`: when you create an instance of this object, it allocates a file handle from the operating system. If you let all references to the stream go out of scope before closing it, what happens to that file handle? The answer is that the stream has a *finalizer* method: a method that's called by the JVM just before the garbage collector reclaims the object. In the case of `FileOutputStream`, the finalizer closes the stream, which releases the file handle back to the operating system — and also flushes any buffers, ensuring that all data is properly written to disk.

Any object can have a finalizer; all you have to do is declare the `finalize()` method:

```
protected void finalize() throws Throwable
{
    // cleanup your object here
}
```

While finalizers seem like an easy way to clean up after yourself, they do have some serious limitations. First, you should never rely on them for anything important, since an object's finalizer may never be called — the application might exit before the object is eligible for garbage collection. There are some other, more subtle problems with finalizers, but I'll hold off on these until we get to phantom references.

Object Life Cycle (without Reference Objects)

Putting it all together, an object's life can be summed up by the simple picture below: it's created, it's used, it becomes eligible for collection, and eventually it's collected. The shaded area represents the time during which the object is "strongly reachable," a term that becomes important by comparison with the reachability provided by reference objects.

Enter Reference Objects

JDK 1.2 introduced the `java.lang.ref` package, and three new stages in the object life cycle: softly-reachable, weakly-reachable, and phantom-reachable. These states only apply to objects eligible for collection — in other words, those with

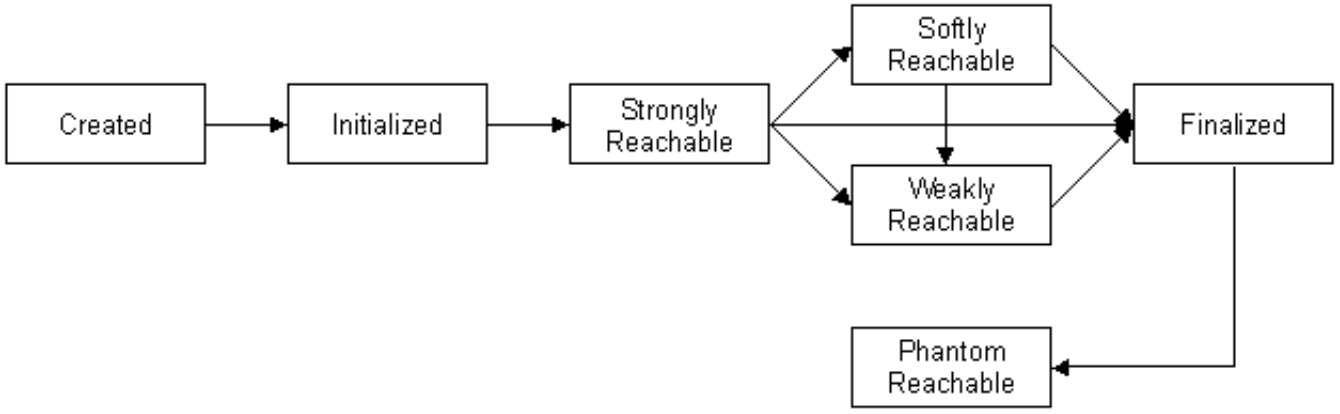
no strong references — and the object in question must be the *referent* of a reference object:

***softly* reachable**
: The object is the referent of a `SoftReference`, and there are no strong references to it. The garbage collector will attempt to preserve the object as long as possible, but will collect it before throwing an `OutOfMemoryError`.

***weakly* reachable**
: The object is the referent of a `WeakReference`, and there are no strong or soft references to it. The garbage collector is free to collect the object at any time, with no attempt to preserve it. In practice, the object will be collected during a major collection, but may survive a minor collection.

***phantom* reachable**
: The object is the referent of a `PhantomReference`, and it has already been selected for collection and its finalizer (if any) has run. The term “reachable” is really a misnomer in this case, as there's no way for you to access the actual object, but it's the terminology that the API docs use.

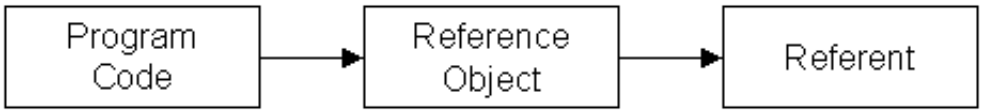
As you might guess, adding three new optional states to the object life-cycle diagram makes for a mess. Although the documentation indicates a logical progression from strongly reachable through soft, weak, and phantom, to reclaimed, the actual progression depends on what reference objects your program creates. If you create a `WeakReference` but don't create a `SoftReference`, then an object progresses directly from strongly-reachable to weakly-reachable to finalized to collected.



It's also important to understand that not all objects are attached to reference objects — in fact, very few of them should be. A reference object is a layer of indirection: you go through the reference object to reach the referent, and clearly you don't want that layer of indirection throughout your code. Most programs, in fact, will use reference objects to access a relatively small number of the objects that the program creates.

References and Referents

A reference object is a layer of indirection between your program code and some other object, called a *referent*. Each reference object is constructed around its referent, and the referent cannot be changed.



The reference object provides the `get()` method to retrieve a strong reference to its referent. The garbage collector may reclaim the referent at any point; once this happens, `get()` returns `null`. To use references properly, you need code like the following:

```
SoftReference<List<Foo>> ref = new SoftReference<List<Foo>>(new LinkedList<Foo>());

// somewhere else in your code, you create a Foo that you want to add to the list
List<Foo> list = ref.get();
if (list != null)
{
    list.add(foo);
}
else
{
    // list is gone; do whatever is appropriate
}
```

Or in words:

1. *You must always check to see if the referent is `null`*
   The garbage collector can clear the reference at any time, and if you blithely use the reference, sooner or later you'll get a `NullPointerException`.
2. *You must hold a strong reference to the referent to use it*
   Again, the garbage collector can clear the reference at *any* time, even in the middle of a single expression. If, rather than creating the `list` variable, I simply call `ref.get().add(foo)`, the reference might be cleared between the

check for null and the actual use. Always remember that the garbage collector runs in its own thread, and doesn't care what your code is doing.

3. *You must hold a strong reference to the reference object*
   If you create a reference object, but allow it to go out of scope, then the reference object itself will be garbage-collected. Seems obvious, but it's easy to forget, especially when you're using reference queues (*qv*) to track your references.

Also remember that soft, weak, and phantom references only come into play when there are no more strong references to the referent. They exist to let you hold onto objects past the point where they'd normally become food for the garbage collector. This may seem like a strange thing — if you no longer hold a strong reference, why would you care about the object? The reason depends on the specific reference type.

Soft References

We'll start to answer that question with soft references. If an object is the referent of a `SoftReference` and there are no strong references to it, then the garbage collector is free to reclaim the object but *tries* not to. As a result, a softly-referenced object may survive garbage collection — even several cycles of garbage collection — as long as the JVM is able to recover "enough" memory without clearing it.

The JDK documentation says that soft references are appropriate for a memory-sensitive cache: each of the cached objects is accessed through a `SoftReference`, and if the JVM decides that it needs space, then it will clear some or all of the references and reclaim their referents. If it doesn't need space, then the referents remain in the heap and can be accessed be program code. In this scenario, the referents are strongly referenced when they're being actively used, softly referenced otherwise. If a soft reference gets cleared, you need to refresh the cache.

To be useful in this role, however, the cached objects need to be pretty large — on the order of several kilobytes each. Useful, perhaps, if you're implementing a fileserver that expects the same files to be retrieved on a regular basis, or have large object graphs that need to be cached. But if your objects are small, then you'll have to clear a lot of them to make a difference, and the reference objects will add overhead to the whole

Memory-Limited Caches Considered Harmful

In my opinion, available memory is the absolute worst way to manage a cache. If you have a small heap you'll be constantly reloading objects, whether or not they're actively used — and never know it, because the cache will silently dispose them. A large heap is worse: you'll hold onto objects long past their natural lifetime. This will slow down your application with every garbage collection, which has to examine those objects. And if the objects aren't accessed otherwise, there's a chance that those sections of the heap will have been swapped out and you'll have lots of page faults

difference, and the reference objects will add overhead to the whole process.

during the collection.

Bottom line: if you use a cache, give thought to how it will be used, and pick a caching strategy (LRU, timed LRU, whatever) that fits that need. And think long and hard before picking a memory-based strategy.

Soft Reference as Circuit Breaker

A better use of soft references is to provide a "circuit breaker" for memory allocation: put a soft reference between your code and the memory it allocates, and you avoid the dreaded `OutOfMemoryError`. This technique works because memory allocation tends to be localized within the application: reading rows from a database, or processing data from a file.

For example, if you write a lot of JDBC code, you might have a method like the following to process query results in a generic way and ensure that the `ResultSet` is properly closed. It only has one small flaw: what happens if the query returns a million rows and you don't have available memory to store them?

```
public static List<List<Object>> processResults(ResultSet rslt)
throws SQLException
{
    try
    {
        List<List<Object>> results = new LinkedList<List<Object>>();
        ResultSetMetaData meta = rslt.getMetaData();
        int colCount = meta.getColumnCount();

        while (rslt.next())
        {
            List<Object> row = new ArrayList<Object>(colCount);
            for (int ii = 1 ; ii <= colCount ; ii++)
                row.add(rslt.getObject(ii));

            results.add(row);
        }

        return results;
    }
    finally
    {
        closeQuietly(rslt);
    }
```

```
}
```

The answer, of course, is an `OutOfMemoryError`. Which makes this the perfect place for a circuit breaker: if the JVM runs out of memory while processing the query, release all the memory that it's already used, and throw an application-specific exception.

At this point, you may wonder: who cares? The query is going to abort in either case, why not just let the out-of-memory error do the job? The answer is that your application may not be the only thing affected. If you're running on an application server, your memory usage could take down other applications. Even in an unshared environment, a circuit-breaker improves the robustness of your application, because it confines the problem and gives you a chance to recover and continue.

To create the circuit breaker, the first thing you need to do is wrap the results list in a `SoftReference` (you've seen this code before):

```
SoftReference<List<List<Object>>> ref
    = new SoftReference<List<List<Object>>>(new LinkedList<List<Object>>());
```

And then, as you iterate through the results, create a strong reference to the list *only when you need to update it*:

```
while (rslt.next())
{
    rowCount++;
    // store the row data

    List<List<Object>> results = ref.get();
    if (results == null)
        throw new TooManyResultsException(rowCount);
    else
        results.add(row);

    results = null;
}
```

This works because almost all of the method's memory allocation happens in two places: the call to `next()`, and the code that stores the row's data in its own list. In the first case, there's a lot that happens when you call `next()`: the `ResultSet` typically retrieves a large block of binary data, containing multiple rows. Then, when you call `getObject()`, it extracts a piece of that data and wraps it in a Java object.

While those expensive operations happen, the only reference to the list is via the `SoftReference`. If you run out of memory the reference will be cleared, and the list will become garbage. It means that the method throws, but the effect of that throw can be confined. And perhaps the calling code can recreate the query with a retrieval limit.

Once the expensive operations complete, you can hold a strong reference to the list with relative impunity. However, note that I use a `LinkedList` for my results rather than an `ArrayList`: I know that linked lists grow in increments of a few dozen bytes, which is unlikely to trigger `OutOfMemoryError`. By comparison, if an `ArrayList` needs to increase its capacity, it must create a new array to do so. In a large list, this could mean a multi-megabyte allocation.

Also note that I set the `results` variable to `null` after adding the new element; this is one of the few cases where doing so is justified. Although the variable goes out of scope at the end of the loop, the garbage collector might not know that (because there's no reason for the JVM to clear the variable's slot in the call stack). So, if I didn't clear the variable, it would be an unintended strong reference during the subsequent pass through the loop.

Soft References Aren't A Silver Bullet

While soft references can prevent many out-of-memory conditions, they can't prevent all of them. The problem is this: in order to actually use a soft reference, you have to create a strong reference to the referent: to add a row to the results, we need to have a reference to the actual list. During the time we hold that string reference, we are at risk for an out-of-memory error.

The goal with a circuit breaker is to minimize the window during which it's useless: the time that you hold a strong reference to the object, and perhaps more important, the amount of allocation that happens during this time. In our case, we confine the strong reference to adding a row to the results, and we use a `LinkedList` rather than an `ArrayList` because the former grows in much smaller increments.

And I want to repeat that, while I hold the strong reference in a variable that quickly goes out of scope, the language spec says nothing about the JVM being required to clear variables that go out of scope. And as-of this writing, the Oracle/OpenJDK JVM does not do so. If I didn't explicitly clear the `results` variable, it would remain a strong reference throughout the loop, acting like a penny in a fuse box, and preventing the soft reference from doing its job.

Finally, think carefully about non-obvious strong references. For example, you might want to add a circuit breaker while

constructing XML documents using the DOM. However, each node in a DOM holds a reference to its parent, in effect holding a reference to every other node in the tree. And if you use a recursive call to build that document, your stack might be full of references to individual nodes.

Weak References

A weak reference, as its name suggests, is a reference object that doesn't put up a fight when the garbage collector comes knocking. If there are no strong or soft references to the referent, it's all but guaranteed to be collected. So what's the use? There are two main uses: associating objects that have no inherent relationship, and reducing duplication via a canonicalizing map.

The Problem With `ObjectOutputStream`

As an example of the first case, I'm going to look at object serialization, which doesn't use weak references. `ObjectOutputStream` and its partner `ObjectInputStream` provide a way to transform arbitrary Java object graphs into a stream of bytes and back again. From the perspective of object modeling, there is no relationship between the streams and the objects written using those streams: the stream is not composed of the objects that are written, nor does it aggregate them.

But when you look at the stream specification, you see that there is in fact a relationship: in order to preserve object identity, the output stream associates a unique identifier with each object written, and subsequent requests to write the object instead write the identifier. This feature is absolutely critical to the stream's ability to serialize arbitrary object graphs: if it wasn't present, a self-referential graph would turn into an infinite stream of bytes.

To implement this feature, both streams need to maintain a strong reference to every object written to the stream. For the programmer who decides to use object streams as an easy way to layer a messaging protocol onto a socket connection, this is a problem: messages are assumed transient, but the streams hold them in memory. Sooner or later, the program runs out of memory (unless the programmer knows to call `reset()` after every message).

Such non-inherent relationships are surprisingly common: they exist whenever the programmer needs to maintain context essential for the use of an object. Sometimes, as in the case of a servlet `Session` object, these associations are managed implicitly by the environment. Sometimes, as in the case of object streams, the associations must be managed explicitly by the programmer. And other times, as in the case of a static `Map` buried deep within the application code, the associations are only discovered when the production server throws an out-of-memory error.

Weak references provide a way to maintain such associations while letting the garbage collector do its work: the weak

reference remains valid only as long as there are also strong references. Returning to the object stream example, if you're using the stream for messaging, the message will be eligible for collection as soon as it's written. On the other hand, a stream used for RMI access to a long-lived data structure would maintain its sense of identity.

Unfortunately, although the object stream protocol was updated with the 1.2 JDK, and weak references were added at the same time, the JDK developers didn't choose to combine them. So remember to call `reset()`.

Eliminating Duplicate Data with Canonicalizing Maps

Object streams notwithstanding, I don't believe there are many cases where you should associate two objects that don't have an inherent relationship. And some of the examples that I've seen, such as Swing listeners that clean up after themselves, seem more like hacks than valid design choices.

In my opinion, the best use of weak references is to implement a canonicalizing map, a mechanism to ensure that only one instance of a value object exists at a time. `String.intern()` is the classic example of such a map: when you intern a string the JVM adds it to a special map that's also used to hold string literals. The reason to do this is *not*, as some people think, in order to make comparisons faster. It's to minimize the amount of memory consumed by duplicated non-literal strings (such as those read from a file or message queue).

A simple canonicalizing map works by using the same object as key and value: you probe the map with an arbitrary instance, and if there's already a value in the map, you return it. If there's no value in the map, you store the instance that was passed in (and return it). Of course, this only works for objects that can be used as map keys. Here's how we might implement `String.intern()` if we weren't worried about memory leaks:

When I originally wrote this article, circa 1998, I presented a canonicalizing map as an alternative to `String.intern()` on the assumption that interned strings would never be cleaned up. I later learned that concern was groundless. More important, as of JDK 8, OpenJDK has eliminated the permanent generation entirely. So there's no need to fear `intern()`, but a canonicalizing map remains useful for objects other than strings.

```
private Map<String,String> _map = new HashMap<String,String>();

public synchronized String intern(String str)
{
    if (_map.containsKey(str))
        return _map.get(str);
    _map.put(str, str);
    return str;
```

```
}
```

This implementation is fine if you have a small number of strings to intern, perhaps within a single method that processes a file. However, let's say that you're writing a long-running application that has to process input from multiple sources, that contain a wide range of strings but still has a high level of duplication. For example, a server that processes uploaded files of postal address data: there will be lots of entries for New York City, not so many for Temperanceville VA. You would want to eliminate duplication of the former, but not hold onto the latter any longer than necessary.

This is where a canonicalizing map with weak reference helps: it allows you to create a canonical instance *only so long as some code in the program is using it*. After the last strong reference disappears, the canonical string will be collected. If the string is seen again later, it becomes the new canonical instance.

To improve our canonicalizer, we can replace `HashMap` with a `WeakHashMap`:

```
private Map<String,WeakReference<String>> _map
    = new WeakHashMap<String,WeakReference<String>>();

public synchronized String intern(String str)
{
    WeakReference<String> ref = _map.get(str);
    String s2 = (ref != null) ? ref.get() : null;
    if (s2 != null)
        return s2;

    _map.put(str, new WeakReference(str));
    return str;
}
```

First thing to notice is that, while the map's key is a `String`, its value is a `WeakReference<String>`. This is because `WeakHashMap` uses weak references for its *keys*, but holds strong references to its values. Since our key and value are the same, the entry would never get collected. By wrapping the entry, we let the GC collect it.

Second, note the process for returning a string: first we retrieve the weak reference. If it exists, then we retrieve the referent. *But we have to check that object as well*. It's possible that the reference is sitting in the map but is already cleared. Only if the referent is not null do we return it; otherwise we consider the passed-in string to be the new canonical version.

Thirdly, note that I've synchronized the `intern()` method. The most likely use for a canonicalizing map is in a multi-threaded environment such as an app-server, and `WeakhashMap` isn't synchronized internally. The synchronization in this example is actually rather naive, and the `intern()` method can become a point of contention. In a real-world implementation, I might use `ConcurrentHashMap`, but the naive approach works better for a tutorial.

Finally, the documentation for `WeakHashMap` is somewhat vague about when entries get removed from the map. It states that “a `WeakHashMap` may behave as though an unknown thread is silently removing entries.” In reality there is no other thread. Instead, the map cleans up whenever it's accessed. To keep track of which entries are no longer valid, it uses a reference queue.

Reference Queues

While testing a reference for `null` lets you know whether its referent has been collected, doing so isn't very efficient &mash; if you have a lot of references, your program will spend most of its time looking for those that have been cleared.

The better solution is a reference queue: you associate a reference with a queue at construction time, and the reference will be put on the queue after it has been cleared. To discover which references have been cleared, you poll the queue. This can be done with a background thread, but it's often simpler to poll the queue at the time you create new references (this is what `WeakHashMap` does).

Reference queues are most often used with phantom references, described below, but can be used with any reference type. The following code is an example with weak references: it creates a bunch of buffers, accessed via a `WeakReference`, and after every creation looks to see what references have been cleared. If you run this code, you'll see long runs of create messages, interspersed with an occasional run of clear messages when the garbage collector runs.

```
public static void main(String[] argv) throws Exception
{
    Set<WeakReference<byte[]>> refs = new HashSet<WeakReference<byte[]>>();
    ReferenceQueue<byte[]> queue = new ReferenceQueue<byte[]>();

    for (int ii = 0 ; ii < 1000 ; ii++)
    {
        WeakReference<byte[]> ref = new WeakReference<byte[]>(new byte[1000000], queue);
        System.err.println(ii + ": created " + ref);
        refs.add(ref);
```

```
            Reference<? extends byte[]> r2;
            while ((r2 = queue.poll()) != null)
            {
                System.err.println("cleared " + r2);
                refs.remove(r2);
            }
        }
    }
```

As always, there are things to note about this code. First, although we're creating `WeakReference` instances, polling the queue gives us a plain `Reference`. This serves to remind you that, once they're enqueued, it no longer matters what type of a reference you're using; the referent has already been cleared.

Second is that *we must hold a strong reference to the reference objects themselves*. The reference object knows about the queue, but the queue doesn't know about the reference until it's enqueued. If we didn't maintain the strong reference to the reference object, it would itself be collected, and never added to the queue. I use a `Set` in this example, and remove the references once they've been cleared (leaving them in the set is a memory leak).

Phantom References

Phantom references differ from soft and weak references in that they're *not* used to access their referents. Instead, their sole purpose is to tell you when their referent has already been collected. While this seems rather pointless, it actually allows you to perform resource cleanup with more flexibility than you get from finalizers.

The Trouble With Finalizers

Back in the description of object life cycle, I mentioned that finalizers have subtle problems that make them unsuitable for cleaning up non-memory resources. There are also a couple of non-subtle problems, that I'll cover here for completeness and then promptly ignore.

- *A finalizer might never be invoked*
  If your program never runs out of available memory, then the garbage collector won't run, and neither will your finalizers. This usually isn't an issue with long-running (eg, server) applications, but short-running

I have more to say about finalizers in this blog post. The short version is that you should rely on `try` / `catch` / `finally` to clean up resources, rather than finalizers *or* phantom references.

programs may finish without ever running garbage collection. And while there is a way to tell the JVM to run finalizers before the program exits, this is unreliable and may conflict with other shutdown hooks.

- *Finalizers can create another strong reference to an object*
  For example, by adding the object to a collection. This essentially resurrects the object, but, as with Stephen King's *Pet Sematary*, the returned object "isn't quite right." In particular, its finalizer *won't* run when the object is again eligible for collection. Perhaps there's a reason that you would use this resurrection trick, but I can't imagine it, and would look very dimly on code that did.

Now that those are out of the way, I believe the real problem with finalizers is that they introduce a gap between the time that the garbage collector first identifies an object for collection and the time that its memory is actually reclaimed, because finalization happens on its own thread, independent of the garbage collector's thread. The JVM is guaranteed to perform a full collection before it returns `OutOfMemoryError`, but if all objects eligible for collection have finalizers, then the collection will have no effect: those objects remain in memory awaiting finalization. Throw in the fact that a standard JVM only has a single thread to handle finalization for all objects, and some long-running finalizers, and you can see where issues might arise.

The following program demonstrates this behavior: each object has a finalizer that sleeps for half a second. Not much time at all, unless you've got thousands of objects to clean up. Every object goes out of scope immediately after it's created, yet at some point you'll run out of memory (if you want to run this example, I recommend using `-Xmx64m` to make the error happen quickly; on my development machine, with a 3 Gb heap, it literally takes minutes to fail).

```
public class SlowFinalizer
{
    public static void main(String[] argv) throws Exception
    {
        while (true)
        {
            Object foo = new SlowFinalizer();
        }
    }

    // some member variables to take up space -- approx 200 bytes
    double a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z;

    // and the finalizer, which does nothing by take time
    protected void finalize() throws Throwable
    {
        try { Thread.sleep(500L); }
```

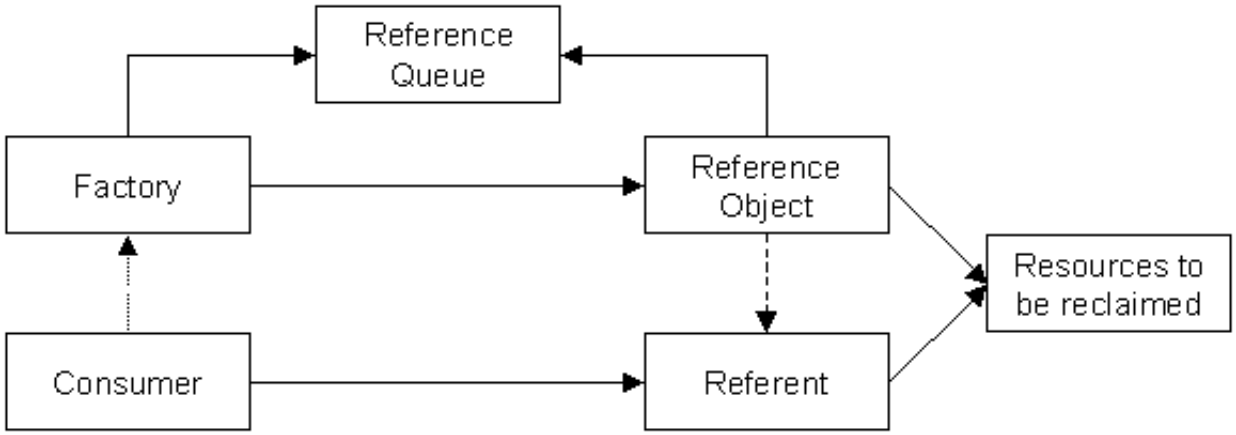
```
            try { Thread.sleep(500L); }
            catch (InterruptedException ignored) {}
            super.finalize();
        }
    }
```

The Phantom Knows

Phantom references allow the application to learn when an object is no longer used, so that the application can clean up the object's non-memory resources. Unlike finalizers, however, the object itself has already been collected by the time the application learns this.

Also unlike finalizers, cleanup is scheduled by the application, not the garbage collector. You might dedicate one or more threads to cleanup, perhaps increasing the number if the number of objects demands it. An alternative — and often simpler — approach is to use an object factory, and clean up after any collected instances before creating a new one.

The key point to understand about phantom references is that *you can't use the reference to access the object*: `get()` always returns `null`, even if the object is still strongly reachable. This means that the referent can't hold the *sole* reference to the resources to be cleaned up. Instead, you must maintain at least one other strong reference to those resources, and use a reference queue to signal that the referent has been collected. As with the other reference types, your program must also hold a strong reference to the reference object itself, or it will be collected and the resources leaked.



Implementing a Connection Pool with Phantom References

Database connections are one of the most precious resources in any application: they take time to establish, and database servers place strict limits on the number of simultaneous open connections that they'll accept. For all that, programmers are remarkably careless with them, sometimes opening a new connection for every query and either forgetting to close it or not closing it in a `finally` block.

Rather than allow the application to open direct connections to the database, most application server deployments use a connection pool: the pool maintains a (normally fixed) set of open connections, and hands them to the program as needed. Production-quality pools provide several ways to prevent connection leaks, including timeouts (to identify queries that run excessively long) and recovery of connections that are left for the garbage collector.

This latter feature serves as a great example of phantom references. To make it work, the `Connection` objects that the pool provides are just wrappers around an actual database connection. They can be collected without losing the database connection because the pool maintains its own strong reference to the actual connection. The pool associates a phantom reference with the "wrapper" connection, and return the actual connection to the pool if and when that reference ends up on a reference queue.

This pool is intended as a demonstration of phantom references, *not* as a production-quality connection pool. There are several production-quality pools available for Java, such as Apache Commons DBCP and C3P0.

The least interesting part of the pool is the `PooledConnection`, shown below. As I said, it's a wrapper that delegates calls to the actual connection. One twist is that I used a reflection proxy for implementation. The JDBC interface has evolved with each version of Java, in ways that are neither forward nor backward compatible; if I had used a concrete implementation, you wouldn't be able to compile the demo unless you used the same JDK version that I did. The reflection proxy solves this problem, and also makes the code quite a bit shorter.

```
public class PooledConnection
implements InvocationHandler
{
    private ConnectionPool _pool;
    private Connection _cxt;

    public PooledConnection(ConnectionPool pool, Connection cxt)
    {
        _pool = pool;
        _cxt = cxt;
    }

    private Connection getConnection()
```

```
private Connection getConnection()
{
    try
    {
        if ((_cxt == null) || _cxt.isClosed())
            throw new RuntimeException("Connection is closed");
    }
    catch (SQLException ex)
    {
        throw new RuntimeException("unable to determine if underlying connection is op
    }

    return _cxt;
}

public static Connection newInstance(ConnectionPool pool, Connection cxt)
{
    return (Connection)Proxy.newProxyInstance(
                PooledConnection.class.getClassLoader(),
                new Class[] { Connection.class },
                new PooledConnection(pool, cxt));
}

@Override
public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable
{
    // if calling close() or isClosed(), invoke our implementation
    // otherwise, invoke the passed method on the delegate
}

private void close() throws SQLException
{
    if (_cxt != null)
    {
        _pool.releaseConnection(_cxt);
        _cxt = null;
    }
}

private boolean isClosed() throws SQLException
{
```

```
    {
        return (_cxt == null) || (_cxt.isClosed());
    }
}
```

The most important thing to note is that `PooledConnection` has a reference to both the underlying database connection and the pool. The latter is used for applications that do remember to close the connection: we want to tell the pool right away, so that the underlying connection can be immediately reused.

The `getConnection()` method also deserves some mention: it exists to catch applications that attempt to use a connection after they've explicitly closed it. This could be a very bad thing if the connection has already been handed to another consumer. So `close()` explicitly clears the reference, and `getConnection()` checks this and throws if the connection is no longer valid. The invocation handler uses this method for all delegated calls.

So now let's turn our attention to the pool itself, starting with the objects it uses to manage connections.

```
private Queue<Connection> _pool = new LinkedList<Connection>();

private ReferenceQueue<Object> _refQueue = new ReferenceQueue<Object>();

private IdentityHashMap<Object,Connection> _ref2Cxt = new IdentityHashMap<Object,Connectic
private IdentityHashMap<Connection,Object> _cxt2Ref = new IdentityHashMap<Connection,Objec
```

Available connections are initialized when the pool is constructed and stored in `_pool`. We use a reference queue, `_refQueue`, to identify connections have been collected. And finally, we have a bidirectional mapping between connections and references, used when returning connections to the pool.

As I've said before, the actual database connection will be wrapped in a `PooledConnection` before it is handed to application code. This happens in the `wrapConnection()` function, which is also where we create the phantom reference and the connection-reference mappings.

```
private synchronized Connection wrapConnection(Connection cxt)
{
```

```
    Connection wrapped = PooledConnection.newInstance(this, cxt);
    PhantomReference<Connection> ref = new PhantomReference<Connection>(wrapped, _refQueue
    _cxt2Ref.put(cxt, ref);
    _ref2Cxt.put(ref, cxt);
    System.err.println("Acquired connection " + cxt );
    return wrapped;
}
```

The counterpart of `wrapConnection` is `releaseConnection()`, and there are two variants of this function. The first is called by `PooledConnection` when the application code explicitly closes the connection. This is the “happy path, ” and it puts the connection back into the pool for later use. It also clears the mappings between connection and reference, as they're no longer needed. Note that this method has default (package) synchronization: it's called by `PooledConnection` so can't be private, but is not generally accessible.

```
synchronized void releaseConnection(Connection cxt)
{
    Object ref = _cxt2Ref.remove(cxt);
    _ref2Cxt.remove(ref);
    _pool.offer(cxt);
    System.err.println("Released connection " + cxt);
}
```

The other variant is called using the phantom reference; it's the “sad path,” followed when the application doesn't remember to close the connection. In this case, all we've got is the phantom reference, and we need to use the mapping to retrieve the actual connection (which is then returned to the pool using the first variant). .

```
private synchronized void releaseConnection(Reference<?> ref)
{
    Connection cxt = _ref2Cxt.remove(ref);
    if (cxt != null)
        releaseConnection(cxt);
}
```

There is one edge case: what happens if the reference gets enqueued after the application has called [illegible]()? This case is

There is one edge case: what happens if the reference gets enqueued after the application has called `close()`? This case is unlikely: when we cleared the mapping, the phantom reference should have become eligible for collection, so it wouldn't be enqueued. However, we have to consider this case, which results in the null check above: if the mapping has already been removed, then the connection has been explicitly returned and we don't need to do anything.

OK, you've seen the low-level code, now it's time for the only method that the application will call:

```
public Connection getConnection()
throws SQLException
{
    while (true)
    {
        synchronized (this)
        {
            if (_pool.size() > 0)
                return wrapConnection(_pool.remove());
        }

        tryWaitingForGarbageCollector();
    }
}
```

The happy path for `getConnection()` that there are connections available in `_pool`. In this case one is removed, wrapped, and returned to the caller. The sad path is that there aren't any connections, in which case the caller expects us to block until until one becomes available. This can happen two ways: either the application closes a connection and it goes back in `_pool`, or the garbage collector finds one that's been abandoned, and enqueues its associated phantom reference.

Before following that path, I want to talk about synchronization. Clearly, all access to the internal data structures must be synchronized, because multiple threads may attempt to get or return connections concurrently. As long as there are connections in `_pool`, the synchronized code executes quickly and the chance of contention is low. However, if we have to loop until connections become available, we want to minimize the amount of time that we're synchronized: we don't want to cause a deadlock between a caller requesting a connection and another caller returning one. Thus the explicit `synchronized` block while checking for connections.

Why am I using `synchronized(this)` rather than an explicit lock? The short answer is that this implementation is intended as a teaching aid, and I want to highlight the synchronization points with minimal boilerplate. In a production-quality pool I would actually avoid explicit

So, what happens if we call `getConnection()` and the pool is empty? This is when we examine the reference queue to find an abandoned connection.

```
private void tryWaitingForGarbageCollector()
{
    try
    {
        Reference<?> ref = _refQueue.remove(100);
        if (ref != null)
            releaseConnection(ref);
    }
    catch (InterruptedException ignored)
    {
        // we have to catch this exception, but it provic
        // a production-quality pool might use it as part
    }
}
```

synchronization entirely, instead relying on concurrent data structures such as [ArrayBlockingQueue](#) and [ConcurrentHashMap](#).

This function highlights another set of conflicting goals: we don't want to waste time if there aren't any enqueued reference, but we also don't want to spin in a tight loop in which we repeatedly check `_pool` and `_refQueue`. So I use a short timeout when polling the queue; it there's nothing there, we'll give another thread the chance to return a connection. This does, of course, introduce a fairness problem: while one thread is waiting on the reference queue, another might return a connection that's immediately grabbed by a third. In theory, the waiting thread could be waiting forever. In the real world, with infrequent need for database connections, this situation is unlikely to happen.

The Trouble with Phantom References

Several pages back, I noted that finalizers are not guaranteed to be called. Neither are phantom references, and for the same reasons: if the collector doesn't run, unreachable objects aren't collected, and references to those objects won't be enqueued. Consider a program did nothing but call `getConnection()` in a loop and let the returned connections go out of scope. If it did nothing else to make the garbage collector run, then it would quickly exhaust the pool and block, waiting for a connection that will never be recovered.

There are, of course, ways to resolve this problem. One of the simplest is to call `System.gc()` in `tryWaitingForGarbageCollector()`. While there is a lot of myth and dogma surrounding this method — the use of the word "suggests" in its documentation has been grist for amateur language lawyers as long as I can remember — it's an

word "suggests" in its documentation has been grist for amateur language lawyers as long as I can remember — it's an effective way to nudge the JVM back toward a desired state. And it's a technique that works for finalizers as well as phantom references.

That doesn't mean that you should ignore phantom references and just use a finalizer. In the case of a connection pool, for example, you might want to explicitly shut down the pool and close all of the underlying connections. You could do that with finalizers, but would need just as much bookkeeping as with phantom references. In that case, the additional control that you get with references (versus an arbitrary finalization thread) makes them a better choice.

A Final Thought: Sometimes You Just Need More Memory

While reference objects are a tremendously useful tool to manage your memory consumption, sometimes they're not sufficient and sometimes they're overkill. For example, let's say that you're building a large object graph, containing data that you read from the database. While you could use soft references as a circuit breaker for the read, and weak references to canonicalize that data, ultimately your program requires a certain amount of memory to run. If you can't give it enough to actually accomplish any work, it doesn't matter how robust your error recovery is.

Your first response to `OutOfMemoryError` should be to figure out why it's happening. Maybe you have a leak, maybe your memory settings are simply too low.

During development, you should specify a large heap size — 1 Gb or more if you have the physical memory - and pay careful attention to how much memory is actually used (`jconsole` is a useful tool here). Most applications will reach a steady state under simulated load, and that should guide your production heap settings . If your memory usage climbs over time, it's quite probable that you are holding strong references to objects after they're no longer in use. Reference objects may help here, but it's more likely that you've got a bug that should be fixed.

The bottom line is that you need to understand your applications. A canonicalizing map won't help you if you don't have duplication. Soft references won't help if you expect to execute multi-million row queries on a regular basis. But in the situations where they can be used, reference objects are often life savers.

Additional Information

You can download the sample programs for this article:

- CircuitBreakerDemo uses a simulated database result-set to trip a memory-driven circuit breaker.

- WeakCanonicalizingMap uses a `WeakHashMap` to create canonical strings. The demo program is perhaps more interesting: it shows the extreme lengths that one has to go to trigger garbage collection (note: running with a large heap is unlikely to work, try using `-Xmx100m`).
- SlowFinalizer show how you can run out of memory even if the garbage collector runs.
- ConnectionPool and PooledConnection implement a very simple connection pool; ConnectionPoolDemo exercises this pool using an in-memory HSQLDB database (here's the Maven POM needed to build this and the other examples).

The “string canonicalizer” class is available from SourceForge, licensed under Apache 2.0.

Sun has many articles on tuning their JVM's memory management. This article is an excellent introduction, and provides links to additional documentation.

Brian Goetz has a great column on the IBM developerWorks site, "Java Theory and Practice." A few years ago, he wrote columns on using both soft and weak references. These articles go into depth on some of the topics that I simply skimmed over, such as using `WeakHashMap` to associate objects with different lifetimes.
