Java's garbage-collected heap

An introduction to the garbage-collected heap of the Java virtual machine

By Bill Venners, JavaWorld.com, 08/01/96

Welcome to another installment of "Under The Hood." This column gives Java developers a glimpse of what is going on underneath their running Java programs. This month's article takes a look at the garbage-collected heap of the Java virtual machine (JVM).

The JVM's heap stores all objects created by an executing Java program. Objects are created by Java's "new" operator, and memory for new objects is allocated on the heap at run time. Garbage collection is the process of automatically freeing objects that are no longer referenced by the program. This frees the programmer from having to keep track of when to free allocated memory, thereby preventing many potential bugs and headaches.

The name "garbage collection" implies that objects that are no longer needed by the program are "garbage" and can be thrown away. A more accurate and up-to-date metaphor might be "memory recycling." When an object is no longer referenced by the program, the heap space it occupies must be recycled so that the space is available for subsequent new objects. The garbage collector must somehow determine which objects are no longer referenced by the program and make available the heap space occupied by such unreferenced objects. In the process of freeing unreferenced objects, the garbage collector must run any finalizers of objects being freed.
In addition to freeing unreferenced objects, a garbage collector may also combat heap fragmentation. Heap fragmentation occurs through the course of normal program execution. New objects are allocated, and unreferenced objects are freed such that free blocks of heap memory are left in between blocks occupied by live objects. Requests to allocate new objects may have to be filled by extending the size of the heap even though there is enough total unused space in the existing heap. This will happen if there is not enough contiguous free heap space available into which the new object will fit. On a virtual memory system, the extra paging required to service an ever growing heap can degrade the performance of the executing program.

This article does not describe an official Java garbage-collected heap, because none exists. The JVM specification says only that the heap of the Java virtual machine must be garbage collected. The specification does not define how the garbage collector must work. The designer of each JVM must decide how to implement the garbage-collected heap. This article describes various garbage collection techniques that have been developed and demonstrates a particular garbage collection technique in an applet.

**Why garbage collection?**

Garbage collection relieves programmers from the burden of freeing allocated memory. Knowing when to explicitly free allocated memory can be very tricky. Giving this job to the JVM has several advantages. First, it can make programmers more productive. When programming in non-garbage-collected languages the programmer can spend many late hours (or days or weeks) chasing down an elusive memory problem. When programming in Java the programmer can use that time more advantageously by getting ahead of schedule or simply going home to have a life.

A second advantage of garbage collection is that it helps ensure program integrity. Garbage collection is an important part of Java's security strategy. Java programmers are unable to accidentally (or purposely) crash the JVM by incorrectly freeing memory.

A potential disadvantage of a garbage-collected heap is that it adds an overhead that can affect program performance. The JVM has to keep track of which objects are being referenced by the executing program, and finalize and free unreferenced objects on the fly. This activity will likely require more CPU time than would have been required if the program explicitly freed unnecessary memory. In addition, programmers in a garbage-collected environment have less control over the scheduling of CPU time devoted to freeing objects that are no longer needed.
Fortunately, very good garbage collection algorithms have been developed, and adequate performance can be achieved for all but the most demanding of applications. Because Java's garbage collector runs in its own thread, it will, in most cases, run transparently alongside the execution of the program. Plus, if a programmer really wants to explicitly request a garbage collection at some point, `System.gc()` or `Runtime.gc()` can be invoked, which will fire off a garbage collection at that time.

The Java programmer must keep in mind that it is the garbage collector that runs finalizers on objects. Because it is not generally possible to predict exactly when unreferenced objects will be garbage collected, it is not possible to predict when object finalizers will be run. Java programmers, therefore, should avoid writing code for which program correctness depends upon the timely finalization of objects. For example, if a finalizer of an un referenced object releases a resource that is needed again later by the program, the resource will not be made available until after the garbage collector has run the object finalizer. If the program needs the resource before the garbage collector has gotten around to finalizing the un referenced object, the program is out of luck.

**Garbage collection algorithms**

A great deal of work has been done in the area of garbage collection algorithms. Many different techniques have been developed that could be applied to a JVM. The garbage-collected heap is one area in which JVM designers can strive to make their JVM better than the competition's.

Any garbage collection algorithm must do two basic things. First, it must detect garbage objects. Second, it must reclaim the heap space used by the garbage objects and make it available to the program. Garbage detection is ordinarily accomplished by defining a set of *roots* and determining *reachability* from the roots. An object is reachable if there is some path of references from the roots by which the executing program can access the object. The roots are always accessible to the program. Any objects that are reachable from the roots are considered live. Objects that are not reachable are considered garbage, because they can no longer affect the future course of program execution.

In a JVM the root set is implementation dependent but would always include any object references in the local variables. In the JVM, all objects reside on the heap. The local variables reside on the Java stack, and each thread of execution has its own stack. Each local variable is either an object reference or a primitive type, such as int, char, or float. Therefore the roots of any JVM garbage-collected heap will include every object reference on every thread's stack. Another source of roots are any object references, such as strings, in the constant pool of loaded classes. The constant pool of a loaded class may reference
strings stored on the heap, such as the class name, superclass name, superinterface names, field names, field signatures, method names, and method signatures.

Any object referred to by a root is reachable and is therefore a live object. Additionally, any objects referred to by a live object are also reachable. The program is able to access any reachable objects, so these objects must remain on the heap. Any objects that are not reachable can be garbage collected because there is no way for the program to access them.

The JVM can be implemented such that the garbage collector knows the difference between a genuine object reference and a primitive type (for example, an int) that appears to be a valid object reference. However, some garbage collectors may choose not to distinguish between genuine object references and look-alikes. Such garbage collectors are called conservative because they may not always free every unreferenced object. Sometimes a garbage object will be wrongly considered to be live by a conservative collector, because an object reference look-alike referred to it. Conservative collectors trade off an increase in garbage collection speed for occasionally not freeing some actual garbage.

Two basic approaches to distinguishing live objects from garbage are reference counting and tracing. Reference counting garbage collectors distinguish live objects from garbage objects by keeping a count for each object on the heap. The count keeps track of the number of references to that object. Tracing garbage collectors, on the other hand, actually trace out the graph of references starting with the root nodes. Objects that are encountered during the trace are marked in some way. After the trace is complete, unmarked objects are known to be unreachable and can be garbage collected.

Reference counting collectors

Reference counting was an early garbage collection strategy; here a reference count is maintained for each object. When an object is first created its reference count is set to one. When any other object or root is assigned a reference to that object, the object's count is incremented. When a reference to an object goes out of scope or is assigned a new value, the object's count is decremented. Any object with a reference count of zero can be garbage collected. When an object is garbage collected, any objects that it refers to has their reference counts decremented. In this way the garbage collection of one object may lead to the subsequent garbage collection of other objects.

An advantage of this scheme is that it can run in small chunks of time closely interwoven with the execution of the program. This characteristic makes it
particularly suitable for real-time environments where the program can't be interrupted for very long. A disadvantage of reference counting is that it does not detect cycles. A cycle is two or more objects that refer to one another, for example, a parent object that has a reference to its child object, which has a reference back to its parent. These objects will never have a reference count of zero even though they may be unreachable by the roots of the executing program. Another disadvantage is the overhead of incrementing and decrementing the reference count each time. Because of these disadvantages, reference counting currently is out of favor. It is more likely that the JVMs you encounter in the real world will use a tracing algorithm in their garbage-collected heaps.

**Tracing collectors**

Tracing garbage collectors trace out the graph of object references starting with the root nodes. Objects that are encountered during the trace are marked in some way. Marking is generally done by either setting flags in the objects themselves or by setting flags in a separate bitmap. After the trace is complete, unmarked objects are known to be unreachable and can be garbage collected.

The basic tracing algorithm is called *mark and sweep*. This name refers to the two phases of the garbage collection process. In the mark phase, the garbage collector traverses the tree of references and marks each object it encounters. In the sweep phase unmarked objects are freed, and the resulting memory is made available to the executing program. In the JVM the sweep phase must include finalization of objects.

Some Java objects have finalizers, others do not. Objects with finalizers that are left unmarked after the sweep phase must be finalized before they are freed. Unmarked objects without finalizers may be freed immediately unless referred to by an unmarked finalizable object. All objects referred to by a finalizable object must remain on the heap until after the object has been finalized.

**Compacting collectors**

Garbage collectors of JVMs will likely have a strategy to combat heap fragmentation. Two strategies commonly used by mark and sweep collectors are *compacting* and *copying*. Both of these approaches move objects on the fly to reduce heap fragmentation. Compacting collectors slide live objects over free memory space toward one end of the heap. In the process the other end of the heap becomes one large contiguous free area. All references to the moved objects are updated to refer to the new location.

Updating references to moved objects is sometimes made simpler by adding a
level of indirection to object references. Instead of referring directly to objects on the heap, object references refer to a table of object handles. The object handles refer to the actual objects on the heap. When an object is moved, only the object handle must be updated with the new location. All references to the object in the executing program will still refer to the updated handle, which did not move. While this approach simplifies the job of heap defragmentation, it adds a performance overhead to every object access.

**Copying collectors**

Copying garbage collectors move all live objects to a new area. As the objects are moved to the new area, they are placed side by side, thus eliminating any free spaces that may have separated them in the old area. The old area is then known to be all free space. The advantage of this approach is that objects can be copied as they are discovered by the traversal from the root nodes. There are no separate mark and sweep phases. Objects are copied to the new area on the fly, and forwarding pointers are left in their old locations. The forwarding pointers allow objects encountered later in the traversal that refer to already copied objects to know the new location of the copied objects.

A common copying collector is called *stop and copy*. In this scheme, the heap is divided into two regions. Only one of the two regions is used at any time. Objects are allocated from one of the regions until all the space in that region has been exhausted. At that point program execution is stopped and the heap is traversed. Live objects are copied to the other region as they are encountered by the traversal. When the stop and copy procedure is finished, program execution resumes. Memory will be allocated from the new heap region until it too runs out of space. At that point the program will once again be stopped. The heap will be traversed and live objects will be copied back to the original region. The cost associated with this approach is that twice as much memory is needed for a given amount of heap space because only half of the available memory is used at any time.

**Heap Of Fish: a garbage-collected heap in action**

The applet below demonstrates a mark and sweep garbage-collected heap that uses compaction. It uses indirect handles to objects instead of direct references to facilitate compaction. It is called Heap Of Fish because the only type of objects stored on the heap for this demonstration are fish objects that are defined as follows:

```java
class YellowFish {
    YellowFish myFriend;
}
class BlueFish {
```
As you can see, there are three classes of fish -- red, blue, and yellow. The red fish is the largest as it has three instance variables. The yellow fish, with only one instance variable, is the smallest fish. The blue fish has two instance variables and is therefore medium-sized.

The instance variables of fish objects are references to other fish objects. BlueFish.myLunch, for example, is a reference to a YellowFish object. In this implementation of a garbage-collected heap, a reference to an object occupies four bytes. Therefore, the size of a RedFish object is 12 bytes, the size of a BlueFish object is eight bytes, and the size of a YellowFish object is four bytes.

A big difference between the Heap Of Fish code and the kind of code likely to be found in a real JVM stems from the fact that Java does not have pointers. The heaps of real world JVMs would use pointers where Heap Of Fish uses array indexes. In the sections that follow I describe some of the structure of the Java code that implements the heap in the applet. If you are curious about how the heap is implemented you can consult the source code for the ultimate level of detail. The heap data structures and behavior are implemented in the applet source as class GCHeap.

Swimming fish
Heap Of Fish has five modes, which are selectable via radio buttons at the bottom left of the applet. When the applet starts it is in swim mode. Swim mode is just a gratuitous animation. The animation is vaguely reminiscent of the familiar image of a big fish about to eat a medium-sized fish, which is about to eat a small fish.

The other four modes -- allocate fish, assign references, garbage collect, and compact heap -- allow you to interact with the heap. You can instantiate new fish objects in the allocate fish mode. The new fish objects go on the heap as all Java objects do. In the assign references mode you can build a network of local variables and fish that refer to other fish. In garbage collect mode, a mark and sweep operation will free any unreferenced fish. The compact heap mode allows you to slide heap objects so that they are side by side at one end of the heap,
leaving all free memory as one large contiguous block at the other end of the heap.

**Allocate fish**
The allocate fish mode shows the two parts that make up the heap, the *object pool* and *handle pool*. The object pool is a contiguous block of memory from which space is taken for new objects. The object pool is structured as a series of memory blocks. Each memory block has a four-byte header which indicates the length of the memory block and whether it is free. The headers are shown in the applet as black horizontal lines in the object pool.

The object pool in Heap Of Fish is implemented as an array of ints. The first header is always at objectPool[0]. The object pool's series of memory blocks can be traversed by hopping from header to header. Each header gives the length of its memory block, which also reveals where the next header is going to be. The header of the next memory block will be the first int immediately following the current memory block. When a new object is allocated, the object pool is traversed until a memory block is encountered with enough space to accommodate the new object. Allocated objects in the object pool are shown as colored bars. YellowFish objects are shown in yellow, BlueFish objects are shown in blue, and RedFish objects are shown in red. Free memory blocks, those that currently contain no fish, are shown in white.

The handle pool in Heap Of Fish is implemented as an array of objects of a class named ObjectHandle. An ObjectHandle contains information about an object, including the vital index into the object pool array. The object pool index functions as a reference to the actual allocated object's instance data in the object pool. The ObjectHandle also reveals information about the class of the fish object. In a real JVM, each allocated object would need to be associated with the information read in from the class file such as the method bytecodes, names of the class, its superclass, any interfaces it implements, its fields, and the type signatures of its methods and fields. In Heap Of Fish, the ObjectHandle associates each allocated object with information such as its class -- whether it is a RedFish, BlueFish, or YellowFish -- and some data used in displaying the fish in the applet user interface.

The handle pool exists to make it easier to defragment the object pool through compaction. References to objects, which can be stored in local variables of a stack or the instance variables of other objects, are not direct indexes into the object pool array. They are instead indexes into the handle pool array. When objects in the object pool are moved for compaction, only the corresponding ObjectHandle must be updated with the object's new object pool array index.

Each handle in the handle pool that refers to a fish object is shown as a
horizontal bar painted the same color as the fish to which it refers. A line connects each handle to its fish object in the object pool. Those handles that are not currently in use are drawn in white.

**Assign references**

The assign references mode allows you to build a network of references between local variables and allocated fish objects. A reference is merely a local or instance variable that contains a valid object reference. There are three local variables which serve as the roots of garbage collection, one for each class of fish. If you do not link any fish to local variables, then all fish will be considered unreachable and freed by the garbage collector.

The assign references mode has three sub-modes -- move fish, link fish, and unlink fish. The sub-mode is selectable via radio buttons at the bottom of the canvas upon which the fish appear. In move fish mode, you can click on a fish and drag it to a new position. You might want to do this so that your links are easier to see or just because you feel like rearranging fish in the sea.

In link fish mode, you can click on a fish or local variable and drag a link to another fish. The fish or local variable you initially drag from will be assigned a reference to the fish you ultimately drop upon. A line will be shown connecting the two items. A line connecting two fish will be drawn between the nose of the fish with the reference to the tail of the referenced fish.

Class YellowFish has only one instance variable, myFriend, which is a reference to a YellowFish object. Therefore, a yellow fish can only be linked to one other yellow fish. When you link two yellow fish the myFriend variable of the "dragged from" fish will be assigned the reference to the "dropped upon" fish. If this action were implemented in Java code, it might look like:

```java
// Fish are allocated somewhere
YellowFish draggedFromFish = new YellowFish();
YellowFish droppedUponFish = new YellowFish();
// Sometime later the assignment takes place
draggedFromFish.myFriend = droppedUponFish;
```

Class BlueFish has two instance variables, BlueFish myFriend and YellowFish myLunch, therefore a blue fish can be linked to one blue fish and one yellow fish. Class RedFish has three instance variables, RedFish myFriend, BlueFish myLunch, and RedFish mySnack. Red fish can therefore link to one instantiation of each class of fish.
In unlink fish mode, you can disconnect fish by moving the cursor over the line connecting two fish. When the cursor is over the line, the line will turn black. If you click a black line the reference will be set to null and the line will disappear.

**Garbage collect**

The garbage collect mode allows you to drive the mark and sweep algorithm. The Step button at the bottom of the canvas takes you through the garbage collection process one step at a time. You can reset the garbage collector at any time by clicking the Reset button. However, once the garbage collector has swept, the freed fish are gone forever. No manner of frantic clicking of the Reset button will bring them back.

The garbage collection process is divided into a mark phase and a sweep phase. During the mark phase, the fish objects on the heap are traversed depth-first starting from the local variables. During the sweep phase, all unmarked fish objects are freed.

At the start of the mark phase, all local variables, fish, and links are shown in white. Each press of the Step button advances the depth-first traversal one more node. The current node of the traversal, either a local variable or a fish, is shown in magenta. As the garbage collector traverses down a branch, fish along the branch are changed from white to gray. Gray indicates the fish has been reached by the traversal, but there may yet be fish further down the branch that have not been reached. Once the terminal node of a branch is reached, the color of the terminal fish is changed to black and the traversal retreats back up the branch. Once all links below a fish have been marked black, that fish is marked black and the traversal returns back the way it came.

At the end of the mark phase, all reachable fish are colored black and any unreachable fish are colored white. The sweep phase then frees the memory occupied by the white fish.

**Compact heap**

The compact heap mode allows you to move one object at a time to one end of the object pool. Each press of the Slide button will move one object. You can see that only the object instance data in the object pool moves; the handle in the handle pool does not move.

The Heap Of Fish applet allows you to allocate new fish objects, link fish, garbage collect, and compact the heap. These activities can be done in any order as much as you please. By playing around with this applet you should be able to get a good idea how a mark and sweep garbage-collected heap works.
There is some text at the bottom of the applet that should help you as you go along. Happy clicking.

**About the author**

Bill Venners is president of Artima Software Company. Through Artima, he does custom software development and consulting. He has been object oriented for five years, primarily working in C++ on MS Windows. Before that he did a lot of C on Unix and assembly language on various microprocessors. He is currently focused on Java and is interested in the new possibilities afforded by Java for enhancing text, especially educational text. He thinks Java is the coolest thing since mountain bikes.