Chapter 9 of Inside the Java Virtual Machine

Garbage Collection

by Bill Venners

The Java virtual machine's heap stores all objects created by a running Java application. Objects are created by the `new`, `newarray`, `anewarray`, and `multianewarray` instructions, but never freed explicitly by the code. Garbage collection is the process of automatically freeing objects that are no longer referenced by the program.

This chapter does not describe an official Java garbage-collected heap, because none exists. As mentioned in earlier chapters, the Java virtual machine specification does not require any particular garbage collection technique. It doesn't even require garbage collection at all. But until infinite memory is invented, most Java virtual machine implementations will likely come with garbage-collected heaps. This chapter describes various garbage collection techniques and explains how garbage collection works in Java virtual machines.

Accompanying this chapter on the CD-ROM is an applet that interactively illustrates the material presented in the chapter. The applet, named *Heap of Fish*, simulates a garbage-collected heap in a Java virtual machine. The simulation--which demonstrates a compacting, mark-and-sweep collector--allows you to interact with the heap as if you were a Java program: you can allocate objects and assign references to variables. The simulation also allows you to interact with the heap as if you were the Java virtual machine: you can drive the processes of garbage collection and heap compaction. At the end of this chapter, you will find a description of this applet and instructions on how to use it.

Why Garbage Collection?

The name "garbage collection" implies that objects 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 can be recycled so that the space is made 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 portions of heap memory are left in between portions 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 (or swapping) required to service an ever growing heap can degrade the performance of the executing program. On an embedded system with low memory, fragmentation could cause the virtual machine to "run out of memory" unnecessarily.

Garbage collection relieves you from the burden of freeing allocated memory. Knowing when to explicitly free allocated memory can be very tricky. Giving this job to the Java virtual machine has several advantages. First, it can make you more productive. When programming in non-garbage-collected languages you can spend many late hours (or days or weeks) chasing down an elusive memory problem. When programming in Java you 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 Java virtual machine by incorrectly freeing memory.

A potential disadvantage of a garbage-collected heap is that it adds an overhead that can affect program performance. The Java virtual machine 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.

Garbage Collection Algorithms

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 the space available again 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.

The root set in a Java virtual machine is implementation dependent, but would always include any object references in the local variables and operand stack of any stack frame and any object references in any class variables. 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 refer to strings stored on the heap, such as the class name, superclass name, superinterface names, field names, field signatures, method names, and method signatures. Another source of roots may be any object references that were passed to native methods that either haven't been "released" by the native method. (Depending upon the native method interface, a native method may be able to release references by simply returning, by explicitly invoking a call back that releases passed references, or some combination of both.) Another potential source of roots is any part of the Java virtual machine's runtime data areas that are allocated from the garbage-collected heap. For example, the class data in the method area itself could be placed on the garbage-collected heap in some implementations, allowing the same garbage collection algorithm that frees objects to detect and unload unreferenced classes.

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 Java virtual machine 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. (One example is an `int` that, if it were interpreted as a native pointer, would point to an object on the heap.) Some garbage collectors, however, 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 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. In this approach, a reference count is maintained for each object on the heap. When an object is first created and a reference to it is assigned to a variable, the object's reference count is set to one. When any other variable 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 have 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 approach is that a reference counting collector 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 is that reference counting does not detect *cycles*: two or more objects that refer to one another. An example of a cycle is a parent object that has a reference to a child object that has a reference back to the 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 of reference counting is the overhead of incrementing and decrementing the reference count each time.

Because of the disadvantages inherent in the reference counting approach, this technique is currently out of favor. It is more likely that the Java virtual machines 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 Java virtual machine, the sweep phase must include finalization of objects.

Compacting Collectors

Garbage collectors of Java virtual machines 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 space 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 the garbage collector to detect references to objects that have already been moved. The garbage collector can then assign the value of the forwarding pointer to the references so they point to the object's new location.

A common copying collector algorithm 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.

You can see a graphical depiction of a garbage-collected heap that uses a stop and copy

algorithm in Figure 9-1. This figure shows nine snapshots of the heap over time. In the first snapshot, the lower half of the heap is unused space. The upper half of the heap is partially filled by objects. That portion of the heap that contains objects is painted with diagonal gray lines. The second snapshot shows that the top half of the heap is gradually being filled up with objects, until it becomes full as shown in the third snapshot.

At that point, the garbage collector stops the program and traces out the graph of live objects starting with the root nodes. It copies each live object it encounters down to the bottom half of the heap, placing each object next to the previously copied object. This process is shown in snapshot four.

Snapshot five shows the heap after the garbage collection has finished. Now the top half of the heap is unused, and the bottom half is partially filled with live objects. The sixth snapshot shows the bottom half is now becoming gradually filled with objects, until it too becomes full in snapshot seven.

Once again, the garbage collector stops the program and traces out the graph of live objects. This time, it copies each live object it encounters up to the top half of the heap, as shown in snapshot eight. Snapshot nine shows the result of the garbage collection: the bottom half is once again unused space and the top half is partially filled with objects. This process repeats again and again as the program executes.

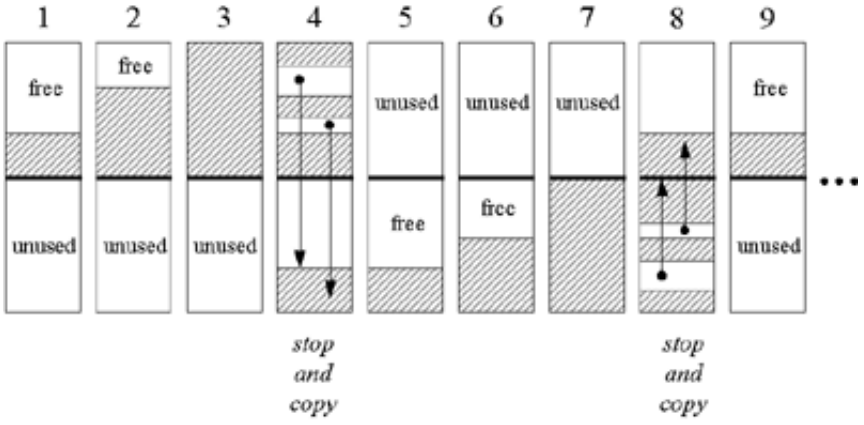

Figure 9-1. A "stop and copy" garbage-collected heap.

Generational Collectors

One disadvantage of simple stop and copy collectors is that *all* live objects must be copied at every collection. This facet of copying algorithms can be improved upon by taking into account two facts that have been empirically observed in most programs in a variety of languages:

1. Most objects created by most programs have very short lives.
2. Most programs create some objects that have very long lifetimes. A major source of

inefficiency in simple copying collectors is that they spend much of their time copying the same long-lived objects again and again.

Generational collectors address this inefficiency by grouping objects by age and garbage collecting younger objects more often than older objects. In this approach, the heap is divided into two or more sub-heaps, each of which serves one "generation" of objects. The youngest generation is garbage collected most often. As most objects are short-lived, only a small percentage of young objects are likely to survive their first collection. Once an object has survived a few garbage collections as a member of the youngest generation, the object is promoted to the next generation: it is moved to another sub-heap. Each progressively older generation is garbage collected less often than the next younger generation. As objects "mature" (survive multiple garbage collections) in their current generation, they are moved to the next older generation.

The generational collection technique can be applied to mark and sweep algorithms as well as copying algorithms. In either case, dividing the heap into generations of objects can help improve the efficiency of the basic underlying garbage collection algorithm.

Adaptive Collectors

An adaptive garbage collection algorithm takes advantage of the fact that some garbage collection algorithms work better in some situations, while others work better in other situations. An adaptive algorithm monitors the current situation on the heap and adjusts its garbage collection technique accordingly. It may tweak the parameters of a single garbage collection algorithm as the program runs. It may switch from one algorithm to another on the fly. Or it may divide the heap into sub-heaps and use different algorithms on different sub-heaps simultaneously.

With an adaptive approach, designers of Java virtual machine implementations need not choose just one garbage collection technique. They can employ many techniques, giving each algorithm work for which it is best suited.

The Train Algorithm

One of the potential disadvantages of garbage collection compared to the explicit freeing of objects is that garbage collection gives programmers less control over the scheduling of CPU time devoted to reclaiming memory. It is in general impossible to predict exactly when (or even if) a garbage collector will be invoked and how long it will take to run. Because garbage collectors usually stop the entire program while seeking and collecting garbage objects, they can cause arbitrarily long pauses at arbitrary times during the execution of the program. Such garbage collection pauses can sometimes be long enough to be noticed by users. Garbage

collection pauses can also prevent programs from responding to events quickly enough to satisfy the requirements of real-time systems. If a garbage collection algorithm is capable of generating pauses lengthy enough to be either noticeable to the user or make the program unsuitable for real-time environments, the algorithm is said to be *disruptive*. To minimize the potential disadvantages of garbage collection compared to the explicit freeing of objects, a common design goal for garbage collection algorithms is to minimize or, if possible, eliminate their disruptive nature.

One approach to achieving (or at least attempting to achieve) non-disruptive garbage collection is to use algorithms that collect incrementally. An *incremental* garbage collector is one that, rather than attempting to find and discard all unreachable objects at each invocation, just attempts to find and discard a portion of the unreachable objects. Because only a portion of the heap is garbage collected at each invocation, each invocation should in theory run in less time. A garbage collector that can perform incremental collections, each of which is guaranteed (or at least very likely) to require less than a certain maximum amount of time, can help make a Java virtual machine suitable for real-time environments. A garbage collector that does its work in time-bounded increments is also desirable in user environments, because such a collector can eliminate garbage collection pauses that are noticeable to the user.

A common incremental collector is a generational collector, which during most invocations collects only part of the heap. As mentioned earlier in this chapter, a generational collector divides the heap into two or more generations, each of which is awarded its own sub-heap. Taking advantage of the empirical observation that most objects have very short lifetimes, a generational collector collects the sub-heaps of younger generations more often than those of older generations. Because every sub-heap except that of the most mature generation (the *mature object space*) can be given a maximum size, a generational collector can in general ensure that incremental collections of all but the most mature generation will complete within a certain maximum amount of time. The reason the mature object space cannot be given a maximum size is that any objects that don't fit in the sub-heaps of the younger generations must by definition go into the mature object space. Such objects have no other place to go.

The train algorithm, which was first proposed by Richard Hudson and Eliot Moss and is currently used by Sun's Hotspot virtual machine, specifies an organization for the mature object space of a generational collector. The purpose of the train algorithm is to provide time-bounded incremental collections of the mature object space.

Cars, Trains, and a Railway Station

The train algorithm divides the mature object space into fixed-sized blocks of memory, each of which is collected individually during a separate invocation of the algorithm. The name, "train algorithm," comes from the way the algorithm organizes the blocks. Each block belongs to one set. The blocks within a set are ordered, and the sets themselves are ordered. To help explain

the algorithm in their original paper, Hudson and Moss called blocks "cars" and sets "trains." In this metaphor, the mature object space plays the role of a railway station. Blocks within the same set are ordered, just like cars within the same train are ordered. The sets are ordered within the mature object space much like trains might line up on track 1, track 2, track 3, and so on, at a railway station. This organization is shown graphically in Figure 9-2.

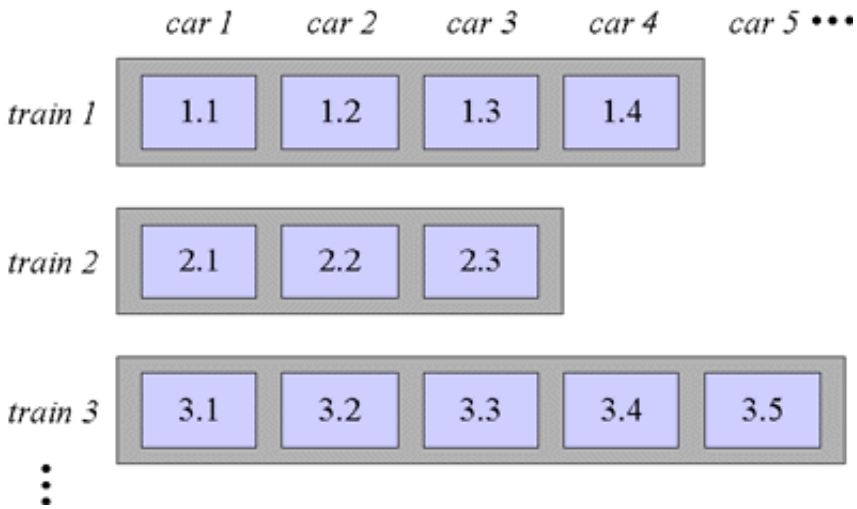

Figure 9-2. Heap organization for the train algorithm.

Trains (sets of blocks) are assigned numbers in the order in which they are created. At the railway station, therefore, the first train to arrive pulls into track 1 and becomes train 1. The next train to arrive pulls into track 2 and becomes train 2. The next train to arrive pulls into track 3 and becomes train 3, and so on. Given this numbering scheme, a smaller train number will always indicate an older train. Within a train, cars (blocks) are added only to the end of the train. The first car added to a train is car 1. The next car added to that same train is car 2. Within a single train, therefore, a smaller car number indicates an older car. This numbering scheme yields an overall order for blocks in the mature object space.

Figure 9-2 shows three trains, numbered 1, 2, and 3. Train 1 has four cars, labeled 1.1 through 1.4. Train 2 has three cars, labeled 2.1 through 2.3. And train 3 has five cars, labeled 3.1 though 3.5. This manner of labeling cars, in which labels are composed of the train number, a dot, and the car number, indicates the overall order of the blocks contained in the mature object space. Car 1.1 precedes car 1.2, which precedes car 1.3, and so on. The last car in train 1 also precedes the first car in train 2, so car 1.4 precedes car 2.1. Likewise, car 2.3 precedes car 3.1. Each time the train algorithm is invoked, it will garbage collect one and only one block: the lowest numbered block. Thus, the first time the train algorithm is invoked on the heap shown in Figure 9-2, it will collect block 1.1. The next time is invoked, it will collect block 1.2. After it collects the last block of train 1, the algorithm will at its next invocation collect the first block of train 2.

Objects arrive in the mature object space when they get old enough to be promoted from the sub-heap of a younger generation. Whenever objects are promoted into the mature object space

from younger generations, they are either added to any existing train except the lowest-numbered train, or one or more new trains are created to hold them. Thus you can think of new objects arriving at the railway station in one of two ways. Either they roll up in cars that are shunted onto the end of any existing train except the lowest numbered train, or they pull into the railway station on a brand new train.

Collecting Cars

Each time the train algorithm is invoked, it collects either the lowest numbered car of the lowest numbered train or it collects the entire lowest numbered train. The algorithm first checks for references into any car of the lowest numbered train. If no references exist outside the lowest numbered train that refer to objects contained inside the lowest numbered train, the entire lowest numbered train contains garbage and can be reclaimed. This first step enables the train algorithm to collect large cyclic data structures that don't fit within a single block. Because of the second step of train algorithm, which will be described next, such large cyclic data structures are guaranteed to eventually end up in the same train.

If the lowest numbered train was determined to all be garbage, the train algorithm reclaims the space occupied by all objects in all the cars of the lowest numbered train and returns. (At that point, that invocation of the train algorithm is complete.) If the lowest numbered train was not all garbage, however, the algorithm turns its attention to the lowest numbered car of the lowest numbered train. In the process, the algorithm will either move or free any object in that car. The algorithm starts by moving any object that is referenced from outside the lowest numbered car to some other car. Any objects remaining in the car after this moving process are unreferenced and can be garbage collected. The train algorithm then reclaims the space occupied by the entire lowest numbered car (thereby freeing any unreferenced objects still sitting in the lowest numbered car) and returns.

The key to guaranteeing that cyclic data structures all end up in the same train lies in how the algorithm moves objects. If an object sitting in the car being collected is referenced from outside the mature object space, that object is moved to any train but the one being collected. If an object is referenced from a different train within the mature object space, that object is moved to the referencing train. Moved objects are then scanned for references back into the car being collected. Any newly referenced objects are moved to the referencing train. Newly moved objects are then scanned for references back into the car being collected, and the process repeats until no more references exist from other trains into the car being collected. If a receiving train runs out of space, the algorithm will create a new car (an empty block) and append it to the end of that train.

Once no more references exist from outside the mature object space or from other trains within the mature object space into the car being collected, any objects referenced from outside the car being collected are known to be referenced from other cars of the same train. The algorithm

moves such objects to the last car of the same, lowest numbered train. These objects are then scanned for references back into the car being collected. Any newly referenced objects are moved to the end of the same train and scanned. This process repeats until no more references of any kind exist into the car being collected. The algorithm then reclaims the space occupied by the entire lowest-numbered car, freeing any unreferenced objects that still happen to be sitting in that car, and returns.

At each invocation, therefore, the train algorithm either collects the lowest numbered car of the lowest numbered train, or it collects the entire lowest numbered train. One of the most important facets of the train algorithm is that it guarantees that large cyclic data structures will eventually be collected, even though they may not fit in a single block. Because objects are moved into trains from which they are referenced, related objects tend to cluster together. Eventually, all the objects of a garbage cyclic data structure, no matter how large, will end up in the same train. Increasing the size of the cyclic data structure will only increase the number of cars that ultimately form the same train. Because the train algorithm first checks for a lowest numbered train that is completely garbage before settling for just the lowest numbered car, it is able to collect cyclic data structures of any size.

Remembered Sets and Popular Objects

As mentioned previously, the goal of the train algorithm is to provide time-bounded incremental collections of the mature object space of a generational collector. Because the blocks (cars) can be given a maximum size and only one block is collected at each invocation, the train algorithm can most often ensure that each invocation will require less than some maximum amount of time. Unfortunately, the train algorithm can't guarantee that each invocation will take less than some maximum amount of time, because the algorithm must do more than just copy the objects.

To facilitate the collection process, the train algorithm makes use of remembered sets. A *remembered set* is a data structure that contains information about all references that reside outside a car or train but point into that car or train. The algorithm maintains one remembered set for each car and each train in the mature object space. The remembered set for a particular car, therefore, contains information about the set of references that refer to (or "remember") the objects in that car. An empty remembered set indicates that the objects contained in the car or train are unreferenced (have been "forgotten") by any objects or variables outside the car or train. Forgotten objects are unreachable and can be garbage collected.

The remembered set is an implementation technique that helps the train algorithm do its work more efficiently. When the train algorithm discovers a car with an empty remembered set, it knows the car contains only garbage and can immediately reclaim all the memory occupied by the car. Likewise, when the train algorithm discovers a train with an empty remembered set, it can immediately reclaim all the memory occupied by the entire train. When the train algorithm

moves an object to a different car or train, the information in the remembered set helps it efficiently update all references to the moved object so that they correctly refer to the objects new location.

Although the amount of bytes the train algorithm may have to copy during one invocation is limited by the size of a block, the amount of work required to move a *popular object*, an object that has many references to it, is impossible to limit. Each time the algorithm moves an object, it must traverse the remembered set of that object and update each reference to that object so that the reference points to the new location. Because the number of references to an object cannot be limited, the amount of time required to update the references to a moved object cannot be limited. Thus, in certain cases the train algorithm may still be disruptive. Nevertheless, despite the degenerative case of popular objects, the train algorithm for the most part does a very good job of collecting the mature object space of a generational garbage collector in an incremental, non-disruptive way.

Finalization

In Java, an object may have a finalizer: a method that the garbage collector must run on the object prior to freeing the object. The potential existence of finalizers complicates the job of any garbage collector in a Java virtual machine.

To add a finalizer to a class, you simply declare a method in that class as follows:

```
// On CD-ROM in file gc/ex2/Example2.java
class Example2 {

    protected void finalize() throws Throwable {
        //...
        super.finalize();
    }
    //...
}
```

A garbage collector must examine all objects it has discovered to be unreferenced to see if any include a `finalize()` method.

Because of finalizers, a garbage collector in the Java virtual machine must perform some extra steps each time it garbage collects. First, the garbage collector must in some way detect unreferenced objects (call this Pass I). Then, it must examine the unreferenced objects it has detected to see if any declare a finalizer. If it has enough time, it may at this point in the garbage collection process finalize all unreferenced objects that declare finalizers.

After executing all finalizers, the garbage collector must once again detect unreferenced objects starting with the root nodes (call this Pass II). This step is needed because finalizers can "resurrect" unreferenced objects and make them referenced again. Finally, the garbage collector can free all objects that were found to be unreferenced in both Passes I and II.

To reduce the time it takes to free up some memory, a garbage collector can optionally insert a step between the detection of unreferenced objects that have finalizers and the running of those finalizers. Once the garbage collector has performed Pass I and found the unreferenced objects that need to be finalized, it can run a miniature trace starting not with the root nodes but with the objects waiting to be finalized. Any objects that are (1) not reachable from the root nodes (those detected during Pass I) and (2) not reachable from the objects waiting to be finalized cannot be resurrected by any finalizer. These objects can be freed immediately.

If an object with a finalizer becomes unreferenced, and its finalizer is run, the garbage collector must in some way ensure that it never runs the finalizer on that object again. If that object is resurrected by its own finalizer or some other object's finalizer and later becomes unreferenced again, the garbage collector must treat it as an object that has no finalizer.

As you program in Java, you 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. As mentioned in Chapter 2, "Platform Independence," you should avoid writing programs for which correctness depends upon the timely finalization of objects. For example, if a finalizer of an unreferenced 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 unreferenced object, the program is out of luck.

The Reachability Lifecycle of Objects

In versions prior to 1.2, every object on the heap is in one of three states from the perspective of the garbage collector: reachable, resurrectable, or unreachable. An object is in the *reachable* state if the garbage collector can "reach" the object by tracing out the graph of object references starting with the root nodes. Every object begins its life in the reachable state, and stays reachable so long as the program maintains at least one reachable reference to the object. As soon as the program releases all references to an object, however, the object becomes resurrectable.

An object is in the *resurrectable* state if it is not currently reachable by tracing the graph of references starting with the root nodes, but could potentially be made reachable again later when the garbage collector executes some finalizer. All objects, not just objects that declare a `finalize()` method, pass through the resurrectable state. As mentioned in the previous section, the finalizer for an object may "resurrect" itself or any other resurrectable object by making the objects reachable again. Because any object in the resurrectable state could potentially be made reachable again by its own or some other object's `finalize()` method, the garbage collector cannot reclaim the memory occupied by a resurrectable object before it makes certain the object won't be brought back to life through the execution of a finalizer. By running

the finalizers of all resurrectable objects that declare a `finalize()` method, the garbage collector will transform the state of all resurrectable objects, either back to the reachable state (for objects that get resurrected), or forward to the unreachable state.

The *unreachable* state indicates not only that an object is no longer reachable, but also that the object cannot be made reachable again through the execution of some finalizer. Unreachable objects can no longer have any affect on the running program. The garbage collector, therefore, is free to reclaim the memory they occupy.

In version 1.2, the three original reachability states -- reachable, resurrectable, and unreachable -- were augmented by three new states: softly, weakly, and phantom reachable. Because these three new states represent three new (progressively weaker) kinds of reachability, the state known simply as "reachable" in versions prior to 1.2 is called "strongly reachable" starting with 1.2. Any object referenced directly from a root node, such as a local variable, is strongly reachable. Likewise, any object referenced directly from an instance variable of a strongly reachable object is strongly reachable.

Reference Objects

The weaker forms of reachability involve an entity that was first introduced in version 1.2: the reference object. A *reference object* encapsulates a reference to some other object, called the *referent*. All reference objects are instances of subclasses of the abstract `java.lang.ref.Reference` class. The family of `Reference` classes, which is shown in Figure 9-3, includes three direct subclasses: `SoftReference`, `WeakReference`, and `PhantomReference`. A `SoftReference` object encapsulates a "soft reference" to a referent object; A `WeakReference` object encapsulates a "weak reference" to a referent object; And a `PhantomReference`, unsurprisingly, encapsulates a "phantom reference" to a referent object. The fundamental difference between a strong reference and its three progressively weaker cousins -- soft, weak, and phantom references -- is that whereas a strong reference prevents its referent from being garbage collected, soft, weak, and phantom references do not.

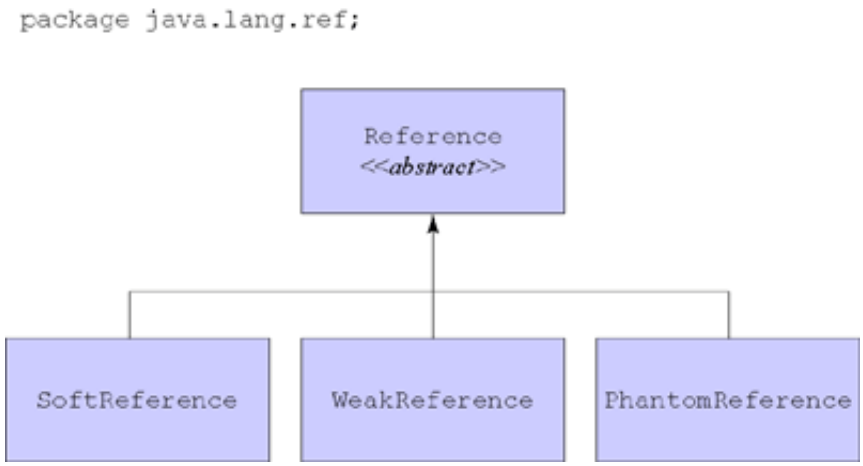

Figure 9-3. The `Reference` family.

To create a soft, weak, or phantom reference, you simply pass a strong reference to the constructor of the appropriate type of reference object. For example, to create a soft reference to a particular `Cow` object, you pass to the constructor of a new `SoftReference` object a strong reference that refers to the `Cow` object. By maintaining a strong reference to the `SoftReference` object, you maintain a soft reference to the `Cow` object.

Figure 9-4 shows such a `SoftReference` object, which encapsulates a soft reference to a `Cow` object. The `SoftReference` object is strongly referenced from a local variable, which, like all local variables, serves as a root node for the garbage collector. As mentioned previously, references contained in garbage collection root nodes and in the instance variables of strongly reachable objects are strong references. Because the `SoftReference` object shown in Figure 9-4 is referenced by a strong reference, the `SoftReference` object is strongly reachable. Assuming that this `SoftReference` object contains the only reference to the `Cow` object, the `Cow` object is softly reachable. The reason the `Cow` is softly reachable is that the garbage collector can only reach the `Cow` object from the root nodes by traversing a soft reference.

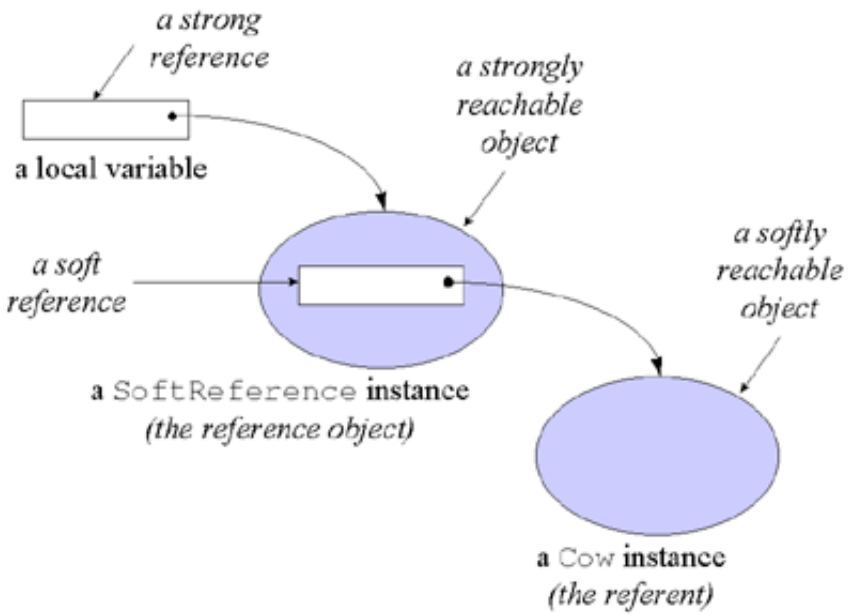

Figure 9-4. A reference object and its referent.

Once a reference object is created, it will continue to hold its soft, weak or phantom reference to its referent until it is *cleared* by the program or the garbage collector. To clear a reference object, the program or garbage collector need only invoke `clear()`, a method defined in class`Reference`, on the reference object. Clearing a reference object invalidates the soft, weak, or phantom reference contained in the reference object. For example, if the program or garbage collector were to invoke `clear()` on the `SoftReference` object shown in Figure 9-4, the soft reference to the `Cow` object would be invalidated, and the `Cow` object would no longer be softly reachable.

Reachability State Changes

As mentioned previously, the purpose of reference objects is to enable you to hold references to objects that the garbage collector is free to collect. Put another way, the garbage collector is allowed to change the reachability state of any object that is not strongly reachable. Because it is often important to keep track of reachability state changes brought about by the garbage collector when you hold soft, weak, or phantom references, you can arrange to be notified when such changes occur. To register interest in reachability state changes, you associate reference objects with reference queues. A *reference queue* is an instance of class `java.lang.ref.ReferenceQueue` to which the garbage collector will append (or "enqueue") reference objects involved in reachability state changes. By setting up and monitoring reference queues, you can stay apprised of interesting reachability state changes performed asynchronously by the garbage collector.

To associate a reference object with a reference queue, you simply pass a reference to the reference queue as a constructor parameter when you create the reference object. A reference object so created, in addition to holding a reference to the referent, will hold a reference to the reference queue. When the garbage collector makes a relevant change to the reachability state of the referent, it will append the reference object to its associated reference queue. For example, when the `WeakReference` object shown in Figure 9-5 was created, two references were passed to the constructor: a reference to a `Fox` object and a reference to a `ReferenceQueue` object. When the garbage collector decides to collect the weakly reachable `Fox` object, it will clear the `WeakReference` object and either at that or some later time append the `WeakReference` object to its reference queue.

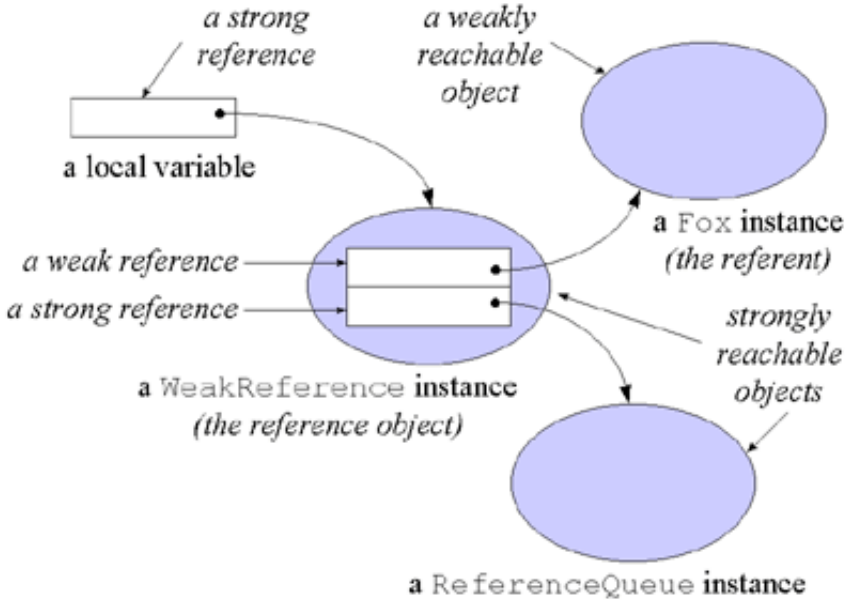

Figure 9-5. A reference object associated with a reference queue.

To append a reference object to the end of its associated queue, the garbage collector invokes

`enqueue()` on the reference object. The `enqueue()` method, which is defined in superclass `Reference`, appends the reference object to a reference queue only if the object was associated with a queue when it was created, and only the first time `enqueue()` is invoked on the object. Programs can monitor a reference queue in two ways, either by polling with the `poll()` method or by blocking with the `remove()` method. If a reference object is waiting in the queue when either `poll()` or `remove()` is invoked on the queue object, the method will remove that object from the reference queue and return it. If no reference object is waiting in the queue, however, `poll()` will immediately return `null`, but `remove()` will block until the next reference object gets enqueued. Once a reference objects arrives in the queue, `remove()` will remove and return it.

The garbage collector enqueues soft, weak, and phantom reference objects in different situations to indicate three different kinds of reachability state changes. The meanings of the six reachability states and the circumstances under which state changes occur are as follow:

- strongly reachable - An object can be reached from the roots without traversing any reference objects. An object begins its lifetime in the strongly reachable state and remains strongly reachable so long as it is reachable via a root node or another strongly reachable object. The garbage collector will not attempt to reclaim the memory occupied by a strongly reachable object.
- softly reachable - An object is not strongly reachable, but can be reached from the roots via one or more (uncleared) soft reference objects. The garbage collector may reclaim the memory occupied by a softly reachable object. If it does so, it clears all soft references to that softly reachable object. When the garbage collector clears a soft reference object that is associated with a reference queue, it enqueues that soft reference object.
- weakly reachable - An object is neither strongly nor softly reachable, but can be reached from the roots via one or more (uncleared) weak reference objects. The garbage collector must reclaim the memory occupied by a weakly reachable object. When it does so, it clears all the weak references to that weakly reachable object. When the garbage collector clears a weak reference object that is associated with a reference queue, it enqueues that weak reference object.
- resurrectable - An object is neither strongly, softly, or weakly reachable, but may still be resurrected back into one of those states by the execution of some finalizer.
- phantom reachable - An object is not strongly, softly, nor weakly reachable, has been determined to not be resurrectable by any finalizer (if it declares a `finalize()` method itself, then its finalizer will have been run), and is reachable from the roots via one or more (uncleared) phantom reference objects. As soon as an object referenced by a phantom reference object becomes phantom reachable, the garbage collector will enqueue it. The garbage collector will never clear a phantom reference. All phantom references must be explicitly cleared by the program.
- unreachable - An object is neither strongly, softly, weakly, nor phantom reachable, and is not resurrectable. Unreachable objects are ready for reclamation.

Note that whereas the garbage collector enqueues soft and weak reference objects when their referents are leaving the relevant reachability state, it enqueues phantom references when the referents are entering the relevant state. You can also see this difference in that the garbage collector clears soft and weak reference objects before enqueueing them, but not phantom reference objects. Thus, the garbage collector enqueues soft reference objects to indicate their referents have just left the softly reachable state. Likewise, the garbage collector enqueues weak reference objects to indicate their referents have just left the weakly reachable state. But the garbage collector enqueues phantom reference objects to indicate their referents have entered the phantom reachable state. Phantom reachable objects will remain phantom reachable until their reference objects are explicitly cleared by the program.

Caches, Canonicalizing Mappings, and Pre-Mortem Cleanup

The garbage collector treats soft, weak, and phantom objects differently because each is intended to provide a different kind of service to the program. Soft references enable you to create in-memory caches that are sensitive to the overall memory needs of the program. Weak references enable you to create canonicalizing mappings, such as a hash table whose keys and values will be removed from the map if they become otherwise unreferenced by the program. Phantom references enable you to establish more flexible pre-mortem cleanup policies than are possible with finalizers.

To use the referent of a soft or weak reference, you invoke `get()` on the reference object. If the reference hasn't been cleared, you'll get a strong reference to the referent, which you can then use in the usual way. If the reference has been cleared, you'll get `null` back. If you invoke `get()` on a phantom reference object, however, you'll always get `null` back, even if the reference object hasn't yet been cleared. Because the phantom reachable state is only attained after an object passes through the resurrectable state, a phantom reference object provides no way to access to its referent. Invoking `get()` on a phantom reference object always returns `null`, even if the phantom reference hasn't yet been cleared, because if it returned a strong reference to the phantom reachable object, it would in effect resurrect the object. Thus, once an object reaches phantom reachability, it cannot be resurrected.

Virtual machine implementations are required to clear soft references before throwing `OutOfMemoryError`, but are otherwise free to decide when or whether to clear them. Implementations are encouraged, however, to clear soft references only when the programs demand for memory exceeds the supply, to clear older soft references before newer ones, and to clear soft references that haven't been used recently before soft references that have been used recently.

Soft references enable you to cache in memory data that you can more slowly retrieve from an external source, such as a file, database, or network. So long as the virtual machine has enough memory to fit the softly referenced data on the heap together with all the strongly referenced

data, the soft reference will in general be strong enough to keep the softly referenced data on the heap. If memory becomes scarce, however, the garbage collector may decide to clear the soft references and reclaim the space occupied by the softly referenced data. The next time the program needs to use that data, it will have to be reloaded from the external source. In the mean time, the virtual machine has more room to accommodate the strongly (and other softly) referenced memory needs of the program.

Weak references are similar to soft references, except that whereas the garbage collector is free to decide whether or not to clear soft references to softly reachable objects, it must clear weak references to weakly reachable objects as soon as it determines the objects are weakly reachable. Weak references enable you to create canonicalizing mappings from keys to values. The `java.util.WeakHashMap` class uses weak references to provide just such a canonicalizing mapping. You can add key-value pairs to a `WeakHashMap` instance via the `put()` method, just like you can to an instance of any class that implements `java.util.Map`. But inside the `WeakHashMap`, the key objects are held via weak reference objects that are associated with a reference queue. If the garbage collector determines that a key object is weakly reachable, it will clear and enqueue any weak reference objects that refer to the key object. The next time the `WeakHashMap` is accessed, it will poll the reference queue and extract all weak reference objects that the garbage collector put there. The `WeakHashMap` will then remove from its mapping any key-value pairs for keys whose weak reference object showed up in the queue. Thus, if you add a key-value pair to a `WeakHashMap`, it will remain there so long as the program doesn't explicitly remove it with the `remove()` method and the garbage collector doesn't determine that the key object is weakly reachable.

Phantom reachability indicates that an object is ready for reclamation. When the garbage collector determines that the referent of a phantom reference object is phantom reachable, it appends the phantom reference object to its associated reference queue. (Unlike soft and weak reference objects, which can optionally be created without associating them with a reference queue, phantom reference objects cannot be instantiated without associating the reference object with a reference queue.) You can use the arrival of a phantom reference in a reference queue to trigger some action that you wish to take at the end of an object's lifetime. Because you can't get a strong reference to a phantom reachable object (the `get()` method always returns `null`), you won't be able to take any action that requires you to have access to the instance variables of the target. Once you have finished the pre-mortem cleanup actions for a phantom reachable object, you must invoke `clear()` on the phantom reference objects that refer to it. Invoking `clear()` on a phantom reference object is the coup de gras for its referent, sending the referent from the phantom reachable state to its final resting place: unreachability.

Heap of Fish:A Simulation

The *Heap of Fish* applet, shown in Figures 9-2 through 9-5, demonstrates a compacting, mark

and sweep, garbage-collected heap. To facilitate compaction, this heap uses indirect handles to objects instead of direct references. It is called *Heap of Fish* because the only type of objects stored on the heap for this demonstration are fish objects, defined as follows:

```
// On CD-ROM in file gc/ex1/YellowFish.java
class YellowFish {

    YellowFish myFriend;
}

// On CD-ROM in file gc/ex1/BlueFish.java
class BlueFish {

    BlueFish myFriend;
    YellowFish myLunch;
}

// On CD-ROM in file gc/ex1/RedFish.java
class RedFish {

    RedFish myFriend;
    BlueFish myLunch;
    YellowFish mySnack;
}
```

As you can see, there are three classes of fish: red, yellow, and blue. 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 the instance data of a `RedFish` object is twelve bytes, a `BlueFish` object is eight bytes, and a `YellowFish` object is four bytes.

Heap of Fish has five modes, which can be selected 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, 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. In the allocate fish mode, you can instantiate new fish objects. 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, shown in Figure 9-6, allows you to allocate new fish objects on the heap.

In this mode you can see 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. It is structured as a series of memory blocks. Each memory block has a four-byte header that indicates the length of the memory block and whether or not it is free. The headers are shown in the applet as black horizontal lines in the object pool.

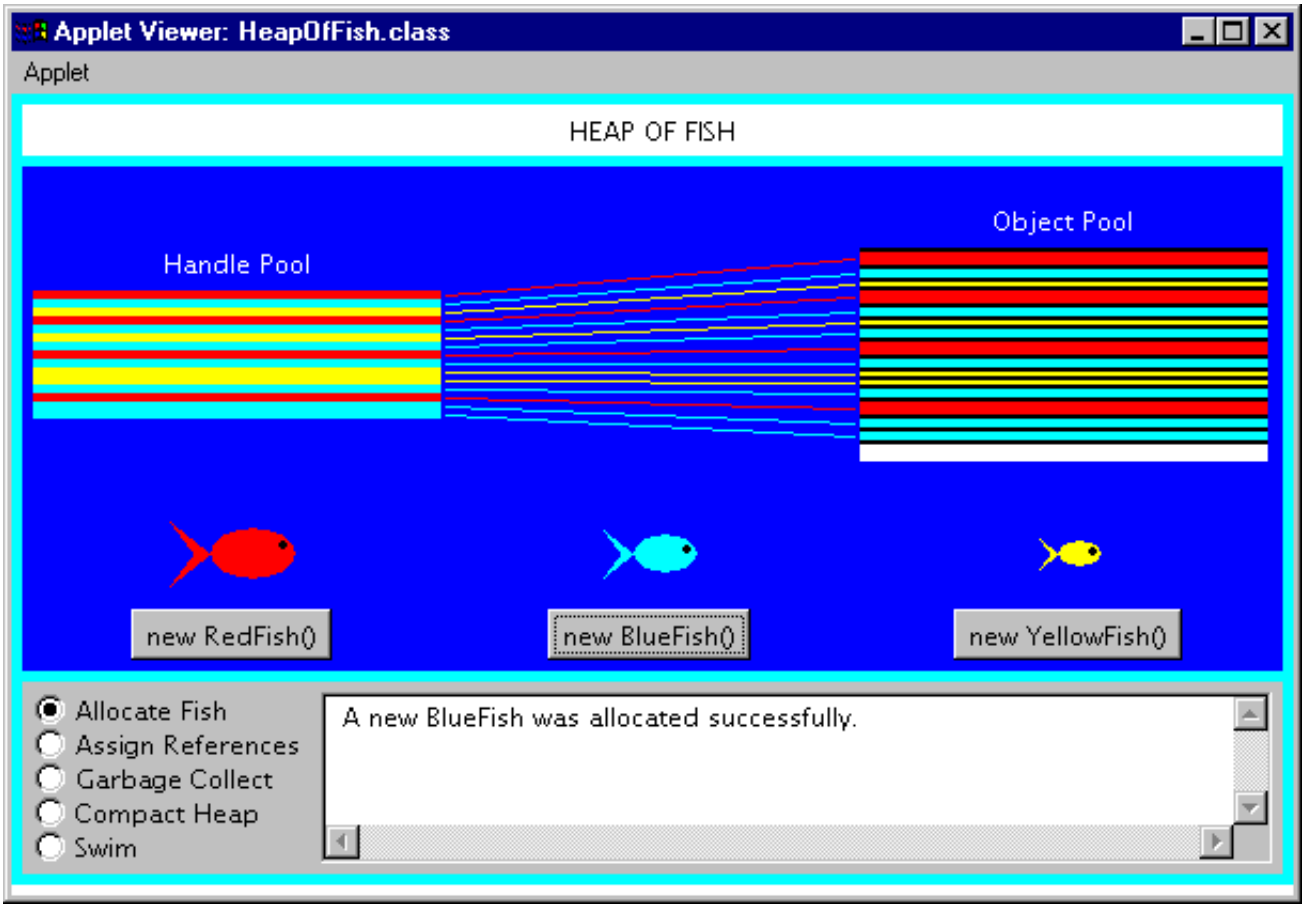

Figure 9-6. The allocate fish mode of the *Heap of Fish* applet.

The object pool in *Heap of Fish* is implemented as an array of `int`s. 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` in blue, and 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 object's

instance data in the object pool. The `ObjectHandle` also reveals information about the class of the fish object. As mentioned in Chapter 5, "The Java Virtual Machine," every object on the heap must in some way be associated with its class information stored in the method area. 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 instance variables in the object pool. Those handles that are not currently in use are drawn in white.

Assign References

The assign references mode, shown in Figure 9-7, 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.

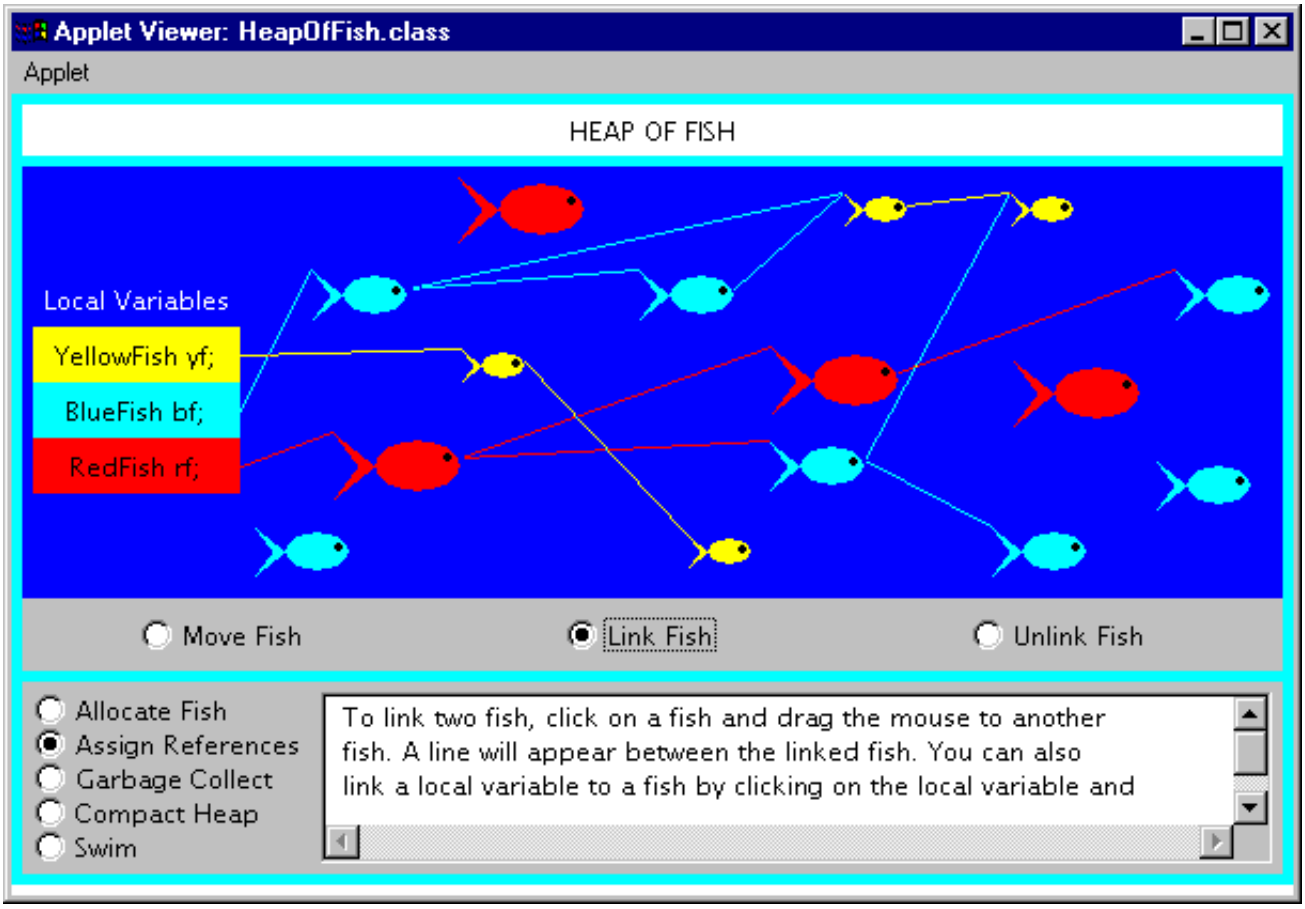

Figure 9-7. The assign references mode of the *Heap of Fish* applet.

The assign references mode has three sub-modes: move fish, link fish, and unlink fish. You can select the sub-mode 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 to make your links more visible 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:

```
// 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 `YellowFish mySnack`. Red fish can therefore link to one instance 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, shown in Figure 9-8, 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.

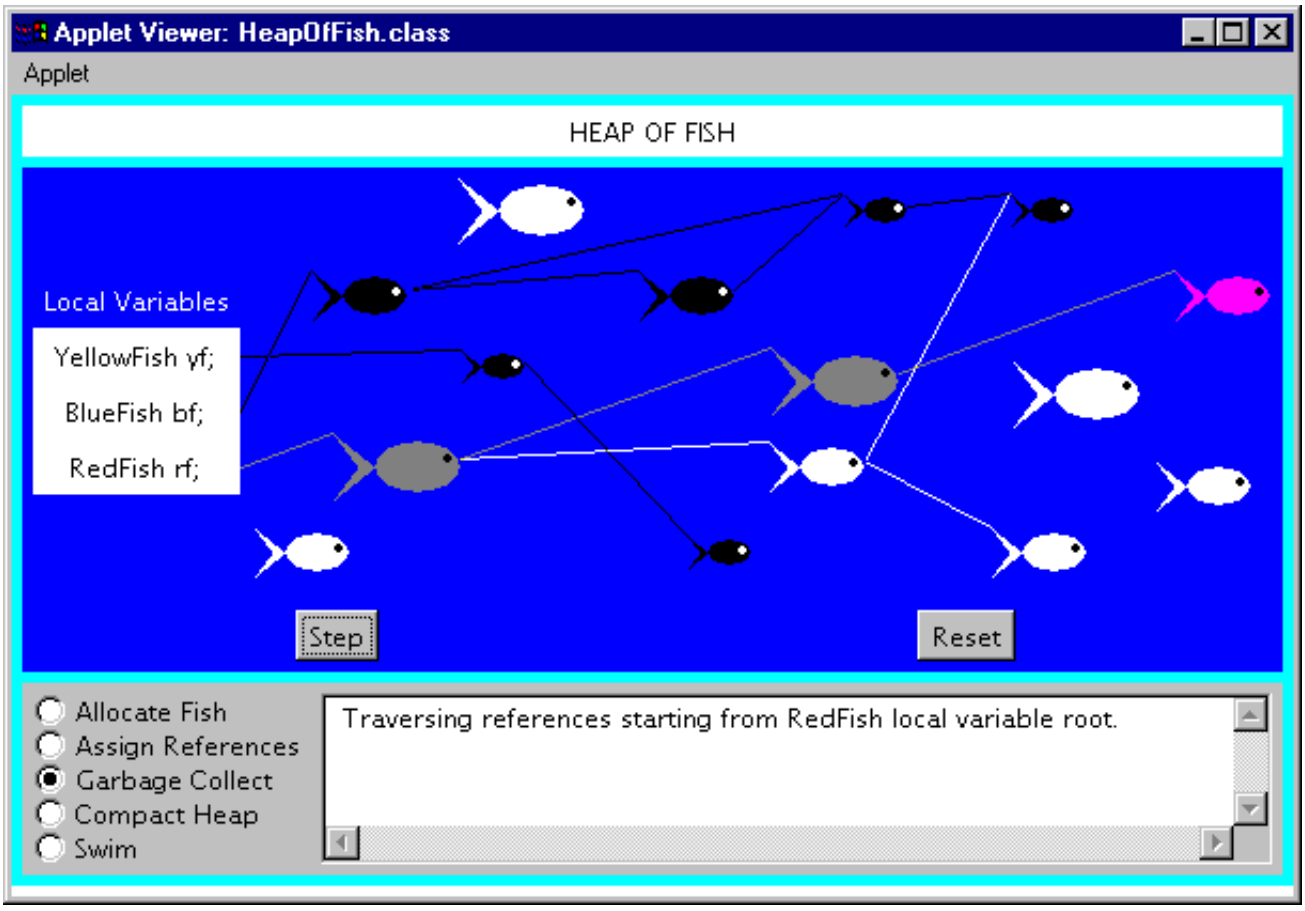

Figure 9-8. The garbage collect mode of the *Heap of Fish* applet.

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, shown in Figure 9-9, 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.

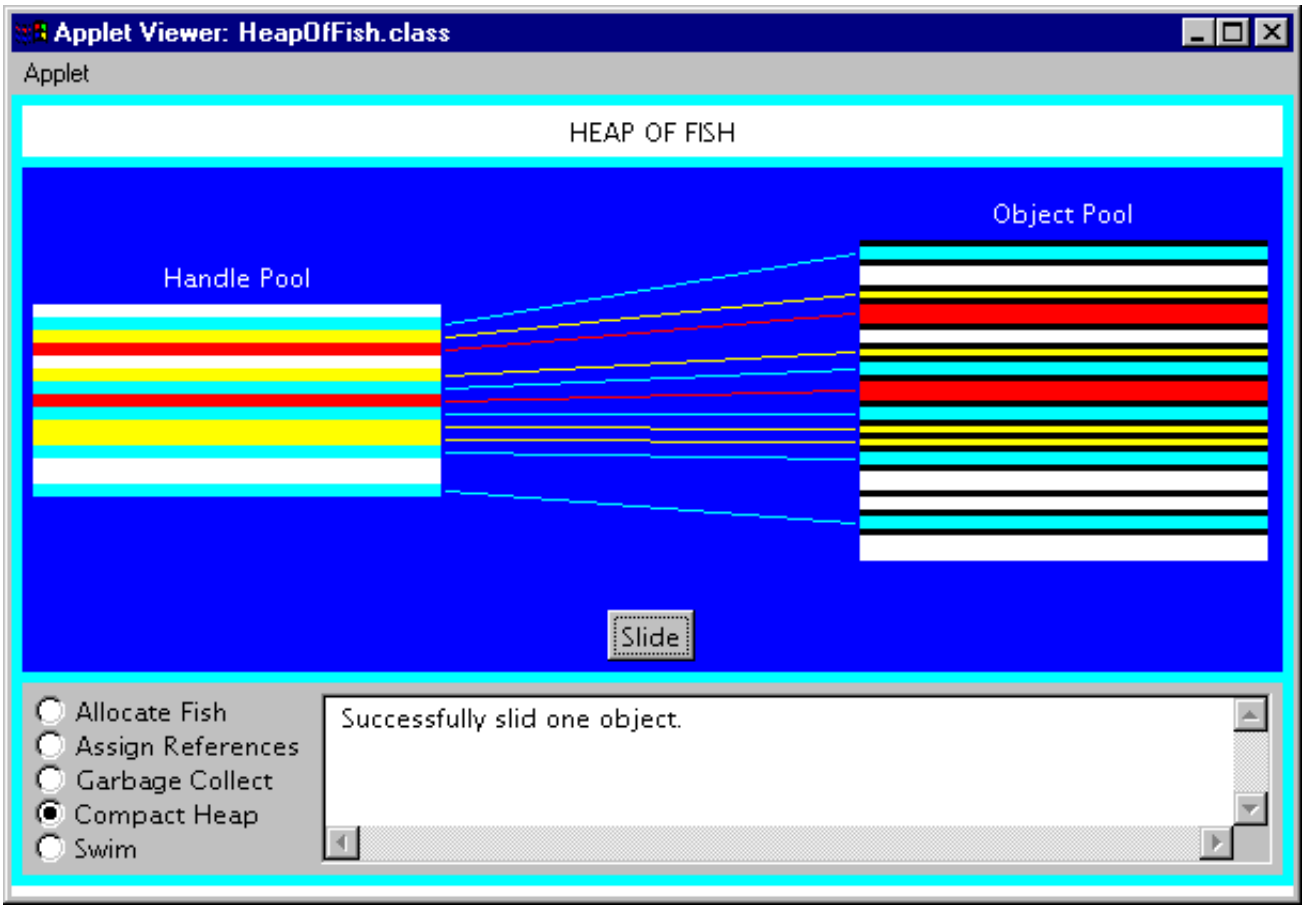

Figure 9-9. The compact heap mode of the *Heap of Fish* applet.

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.

On the CD-ROM

The CD-ROM contains the source code examples from this chapter in the gc directory. The *Heap of Fish* applet is contained in a web page on the CD-ROM in file `applets/HeapOfFish.html`. The source code for this applet is found alongside its class files, in the `applets/HeapOfFish` directory.

The Resources Page

For more information about the material presented in this chapter, visit the resources page: `http://www.artima.com/insidejvm/resources/`.