**The Object Lifecycle**

In order to discuss garbage collection, it is first useful to examine the object lifecycle. An object typically goes through most of the following states between the time it is allocated and the time its resources are finally returned to the system for reuse.

1. Created
2. In use (strongly reachable)
3. Invisible
4. Unreachable
5. Collected
6. Finalized
7. Deallocated

**A.3.1 Created**

When an object is created, several things occur:[2](http://java.sun.com/docs/books/performance/1st_edition/html/JPAppGC.fm.html" \l "997381)

1. Space is allocated for the object.
2. Object construction begins.
3. The superclass constructor is called.
4. Instance initializers and instance variable initializers are run.
5. The rest of constructor body is executed.

The exact costs of these operations depend on the implementation of the JVM, as well as the implementation of the class being constructed. The thing to keep in mind is that these costs exist. Once the object has been created, assuming it is assigned to some variable, it moves directly to the in use state.

**A.3.2 In Use**

Objects that are held by at least one strong reference are considered to be *in use*. In JDK 1.1.x, all references are strong references. Java 2 introduces three other kinds of references: weak, soft and phantom. (These reference types are discussed in [Section A.4.1](http://java.sun.com/docs/books/performance/1st_edition/html/JPAppGC.fm.html#997476).) The example shown in [Listing A-1](http://java.sun.com/docs/books/performance/1st_edition/html/JPAppGC.fm.html#997404) creates an object and assigns it to some variables.

public class CatTest {

static Vector catList = new Vector();

static void makeCat() {

Object cat = new Cat();

catList.addElement(cat);

}

public static void main(String[] arg) {

makeCat();

// do more stuff

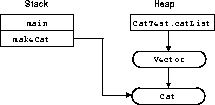
}

}

*Creating and referencing an object*

[Figure A-1](http://java.sun.com/docs/books/performance/1st_edition/html/JPAppGC.fm.html" \l "1004166) shows the structure of the objects inside the VM just before the makeCat method returns. At that moment, two strong references point to the Cat object.

*Object reference graph*



When the makeCat method returns, the stack frame for that method and any temporary variables it declares are removed. This leaves the Cat object with just a single reference from the catList static variable (indirectly via the Vector).

**A.3.3 Invisible**

An object is in the *invisible* state when there are no longer any strong references that are accessible to the program, even though there might still be references. Not all objects go through this state, and it has been a source of confusion for some developers. [Listing A-2](http://java.sun.com/docs/books/performance/1st_edition/html/JPAppGC.fm.html#997426) shows a code fragment that creates an invisible object.

public void run() {

try {

Object foo = new Object();

foo.doSomething();

} catch (Exception e) {

// whatever

}

while (true) { // do stuff } // loop forever

}

*Invisible object*

In this example, the object foo falls out of scope when the try block finishes. It might seem that the foo temporary reference variable would be pulled off the stack at this point and the associated object would become unreachable. After all, once the try block finishes, there is no syntax defined that would allow the program to access the object again. However, an efficient implementation of the JVM is unlikely to zero the reference when it goes out of scope. The object referenced by foo continues to be strongly referenced, at least until the run method returns. In this case, that might not happen for a long time. Because invisible objects can't be collected, this is a possible cause of memory leaks. If you run into this situation, you might have to explicitly null your references to enable garbage collection.

**A.3.4 Unreachable**

An object enters an *unreachable* state when no more strong references to it exist. When an object is unreachable, it is a *candidate* for collection. Note the wording: Just because an object is a candidate for collection doesn't mean it will be immediately collected. The JVM is free to delay collection until there is an immediate need for the memory being consumed by the object.

It's important to note that not just any strong reference will hold an object in memory. These must be references that chain from a garbage collection root. GC roots are a special class of variable that includes

* Temporary variables on the stack (of any thread)
* Static variables (from any class)
* Special references from JNI native code

Circular strong references don't necessarily cause memory leaks. Consider the code in [Listing A-3](http://java.sun.com/docs/books/performance/1st_edition/html/JPAppGC.fm.html#997442). It creates two objects, and assigns them references to each other.

public void buidDog() {

Dog newDog = new Dog();

Tail newTail = new Tail();

newDog.tail = newTail;

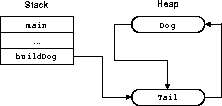
newTail.dog = newDog;

}

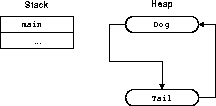
*Circular reference*

[Figure A-2](http://java.sun.com/docs/books/performance/1st_edition/html/JPAppGC.fm.html" \l "1005723) shows the reference graph for the objects before the buildDog method returns. Before the method returns, there are strong references from the temporary stack variables in the buildDog method pointing to both the Dog and the Tail.

*Reference graph before buildDog returns*



*Reference graph after buildDog returns*



[Figure A-3](http://java.sun.com/docs/books/performance/1st_edition/html/JPAppGC.fm.html" \l "1005806) shows the graph for the objects after the buildDog method returns. At this point, the Dog and Tail both become unreachable from a root and are candidates for collection (although the VM might not actually collect these objects for an indefinite amount of time).

**A.3.5 Collected**

An object is in the *collected* state when the garbage collector has recognized an object as unreachable and readies it for final processing as a precursor to deallocation. If the object has a finalize method, then it is marked for finalization. If it does not have a finalizer then it moves straight to the finalized state.

If a class defines a finalizer, then any instance of that class must have the finalizer called prior to deallocation. This means that deallocation is delayed by the inclusion of a finalizer.

**A.3.6 Finalized**

An object is in the *finalized* state if it is still unreachable after its finalize method, if any, has been run. A finalized object is awaiting deallocation. Note that the VM implementation controls when the finalizer is run. The only thing that can be said for certain is that adding a finalizer will extend the lifetime of an object. This means that adding finalizers to objects that you intend to be short-lived is a bad idea. You are almost always better off doing your own cleanup instead of relying on a finalizer. Using a finalizer can also leave behind critical resources that won't be recovered for an indeterminate amount of time. If you are considering using a finalizer to ensure that important resources are freed in a timely manner, you might want to reconsider.

One case where a finalize method delayed GC was discovered by the quality assurance (QA) team working on Swing. The QA team created a stress testing application that simulated user input by using a thread to send artificial events to the GUI. Running on one version of the toolkit, the application reported an OutOfMemoryError after just a few minutes of testing. The problem was finally traced back to the fact that the thread sending the events was running at a higher priority than the finalizer thread. The program ran out of memory because about 10,000 Graphics objects were held in the finalizer queue waiting for a chance to run their finalizers. It turned out that these Graphics objects were holding onto fairly substantial native resources. The problem was fixed by assuring that whenever Swing is done with a Graphics object, dispose is called to ensure that the native resources are freed as soon as possible.

In addition to lengthening object lifetimes, finalize methods can increase object size. For example, some JVMs, such as the classic JVM implementation, add an extra hidden field to objects with finalize methods so that they can be held in a linked list finalization queue.

**A.3.7 Deallocated**

The deallocated state is the final step in garbage collection. If an object is still unreachable after all the above work has occurred, then it is a candidate for deallocation. Again, when and how deallocation occurs is up to the JVM.

**A.4 Reference Objects**

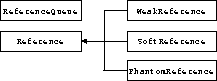
Prior to the introduction of the Java 2 platform, all references were strong references. This meant that there was no way for the developer to interact with the garbage collector, except through brute force methods such as System.gc.

The java.lang.ref package was introduced as part of Java 2. [Figure A-4](http://java.sun.com/docs/books/performance/1st_edition/html/JPAppGC.fm.html#1006000) shows the class hierarchy for the classes in this package. This package defines reference-object classes that enable a limited degree of interaction with the garbage collector. Reference objects are used to maintain a reference to some other object in such a way that the collector can still reclaim the target object. As you might expect, the addition of these new reference objects complicates the concept of reachability as defined in the object lifecycle. Understanding this is important, even if you don't intend to make direct use of this package. Some of the core class libraries use WeakReferences internally, so you might encounter them while using memory profilers to track memory usage.

|  |
| --- |
| Resurrection  It is possible to create new strong references to an object while executing the finalizer method. This puts the object back into an in-use state. This practice, known as *resurrection*, is a bad idea. The specification guarantees that a finalizer is run at most one time per object. Because the finalizer is not run a second time, resurrecting an object can lead to serious problems.  For more information about resurrection, see Ken Arnold and James Gosling, *The Java Programming Language*, Section 2.10.2. Addison-Wesley, 1998. |

|  |
| --- |
|  |

*Reference class hierarchy*



**A.4.1 Types of Reference Objects**

Three types of reference objects are provided, each weaker than the last: soft, weak, and phantom. Each type corresponds to a different level of reachability:

* Soft references are for implementing memory-sensitive caches.
* Weak references are for implementing mappings that do not prevent their keys (or values) from being reclaimed.
* Phantom references are for scheduling pre-mortem cleanup actions in a more flexible way than is possible with the Java finalization mechanism.

Going from strongest to weakest, the different levels of reachability reflect the lifecycle of an object:

* An object is strongly reachable if some thread can reach it without traversing any reference objects.
* An object is softly reachable if it is not strongly reachable but can be reached by traversing a soft reference.
* An object is weakly reachable if it is neither strongly nor softly reachable but can be reached by traversing a weak reference. When the weak references to a weakly reachable object are cleared, the object becomes eligible for finalization.
* An object is phantom reachable if it is neither strongly, softly, nor weakly reachable, it has been finalized, and some phantom reference refers to it.
* An object is unreachable, and therefore eligible for reclamation, when it is not reachable in any of the preceding ways.

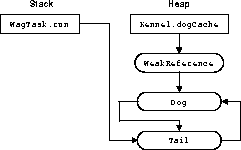
**A.4.2 Example GC with WeakReference**

You're likely to encounter special reference objects while using tools to look for memory leaks. Only strong references will directly interfere with garbage collection. If you find chains of objects linked by weak references, you should be able to ignore them from a GC perspective. (For additional information on the use of special reference objects, see the API documentation.)

[Figure A-5](http://java.sun.com/docs/books/performance/1st_edition/html/JPAppGC.fm.html" \l "1006018) shows a graph of objects in memory for a sample program. Let's say that the problem with this program is that the Dog objects are not being collected, leading to a memory leak. By using a memory profiler, you can find all the pointers to the Dog object and follow them back to their GC roots. There are two GC roots in [Figure A-5](http://java.sun.com/docs/books/performance/1st_edition/html/JPAppGC.fm.html#1006018), a static variable in class Kennel and a stack frame in a live thread. In this case, the WagTask thread is in an infinite loop, forcing the dog's tail to wag. The question is how to get rid of the Dog object.

There are two references pointing to the Dog object, but only one of them is interesting from a GC perspective. The WeakReference from the dogCache is not important. The interesting reference is the reference from the Tail, which chains from a stack frame in a live thread. To free the Dog, and the associated Tail, you need to terminate the thread that is wagging the Tail. Once this thread is gone, everything falls into place. When an object that is pointed to by a WeakReference is collected, the WeakReference is automatically set to null. [Figure A-6](http://java.sun.com/docs/books/performance/1st_edition/html/JPAppGC.fm.html#1005521) shows the result of terminating the wag thread.

*Reference graph*



When the thread dies, its stack is removed. Now the only strong reference to the Dog is via the Tail, and this becomes a simple circular reference that isn't reachable from a GC root. The Dog, and by extension the Tail, are no longer strongly reachable through any references. They are only weakly reachable through the dogCache. When the collector discovers this (which it does on its own schedule), it might set the weak reference to null, making the Dog and Tail totally unreachable. They then become candidates for collection and will be removed at the collector's discretion.

*Results of garbage collection*

