Reference Objects and Garbage Collection

The Reference Object application programming interface (API) enables a program to maintain special references to objects that allow the program to interact with the garbage collector in limited ways. Not all programs require this level of interaction with the garbage collector. For example, a program that keeps a lot of objects in memory or a program that needs to perform cleanup operations on related objects before an object is reclaimed might be good candidates for using the Reference Objects API.

- A web-based program might display a lot of images when an end user goes to a particular page on the web. If the end user leaves the page, it is not always certain he or she will return. Such a program can use reference objects to create a situation where the garbage collector reclaims the images when heap memory runs low. In the event the end user returns to the page, the program can reload the image if it has been reclaimed.
- A program can use a reference queue to create a situation where the program is notified when a certain object is reachable only through reference objects. Upon notification, the program can proceed with clean-up operations on other related objects to make them eligible for garbage collection at the same time.

Because it is essential to understand garbage collection before you can understand how reference objects work, this article begins by describing how garbage collection works when no reference objects are involved, followed by a discussion of how things change when reference objects are added to the heap.

Vanilla Garbage Collection

The garbage collector's job is to identify objects that are no longer in use and reclaim the memory. What does it mean for an object to be *in use*?

> **Note**: An object is in use if it can be accessed or *reached* by the program in its current state.

An executing Java program consists of a set of threads, each of which is actively executing a set of methods (one having called the next). Each of these methods can have arguments or local variables that are references to objects. These references are said to belong to a *root set* of references that are immediately accessible to the program. Other references in the root set include static reference variables defined in loaded classes, and references registered through the Java Native Interface (JNI) API.

All objects referenced by this root set of references are said to be *reachable* by the program in its current state and must not be collected. Also, those objects might contain references to still other objects, which are also reachable, and so on.

All other objects on the heap are considered *unreachable*, and all unreachable objects are eligible for garbage collection. If an unreachable object has a finalize() method, arrangements are made for the object's finalizer to be called. Unreachable objects with no finalizers and objects whose finalizers have been called are simply reclaimed during garbage collection.

Garbage collection algorithms vary, but they all have in common the task of identifying the objects that are reachable from the root set and reclaiming the space occupied by any other objects.

In the diagram, which is a simplified view of the stack and heap for instructional purposes, objects inside the blue square are reachable from the thread root set, while objects outside the square (in red) are not.

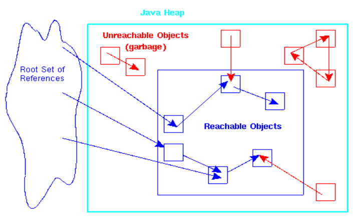

Fig. 1: How garbage collection works.

An object may refer to reachable objects and still be unreachable itself. Likewise, an object can be unreachable in spite of references to it, if those references are all from unreachable objects.

> **Conservative Garbage Collection:** If you use the Java Native Interface (JNI) API to make C calls, the garbage collector might see something in memory created by the C code that looks like a pointer, but is actually garbage. In this case, the memory is not garbage collected because the Java VM is conservative and does not reclaim memory that looks like it could be allocated to a pointer.

What Happens When You Use Reference Objects

How does the introduction of reference objects change things? Unlike ordinary references, the reference in a reference object is treated specially by the garbage collector. A reference object encapsulates a reference to some other object, which is called the *referent*.

The referent of a reference object is specified when the reference object is created. In the following code, `image` is an image object passed to `sr`, a SoftReference object. The `image` object is the referent of `sr`, and the reference field in `sr` is a soft reference to `image`.

```
Image image = (
  sr == null) ? null : (Image)(sr.get());
if (image == null) {
  image = getImage(getCodeBase(), "truck1.gif");
  sr = new SoftReference(image);
}
```

Object Reachability

Besides strongly reachable and unreachable objects, the Reference Objects API gives you strengths of object reachability. You can have softly, weakly and phantomly reachable objects and gain a limited amount of interaction with the garbage collector according to the strength of the object's reachability.

The next sections explain these strengths and how to use them. This section takes a look at how garbage collection is changed when two of the objects on the heap in Figure 1 are changed to weak reference objects.

When an object is reachable from the root set by some chain of ordinary references (no reference objects involved), that object is said to be *strongly reachable*. If, however, the only ways to reach an object involve at least one weak reference object, the object is said to be *weakly reachable*. An object is considered by the garbage collector to be in use if it is strongly reachable. Weakly reachable objects, like unreachable objects, are eligible for collection.

So, weak references allow you to refer to an object without keeping it from being collected. If the garbage collector collects a weakly reachable object, all weak references to it are set to `null` so the object can no longer be accessed through the weak reference.

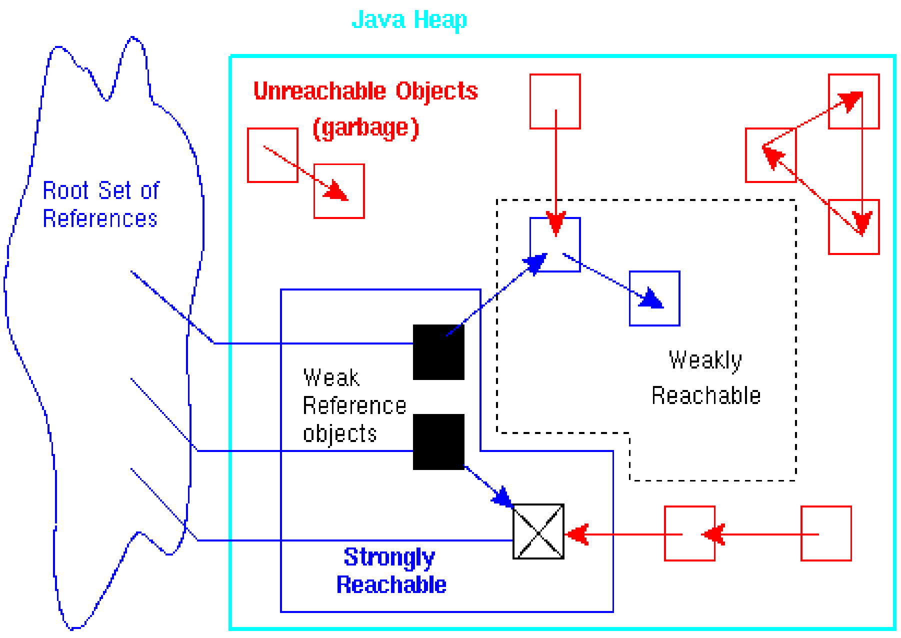

Fig. 2: Adding reference objects to the heap.

- Unreachable objects (outside the strongly and weakly reachable areas) are not reachable from the root set.
- Strongly reachable objects (inside the strongly reachable area to the lower left) are reachable through at least one path that does not go through a reference object.
- Weakly reachable objects (inside the weakly reachable area shown enclosed with a dashed line to the upper-right) are not strongly reachable through any path, but reachable through at least one path that goes through a weak

reference.

The diagram above shows two paths to the object on the heap with the *X*. It can be reached through the weak reference, and directly from the stack without going through a weak reference object. The object with the *X* is strongly reachable and not weakly reachable because one path leads to it from the root set without going through a reference object. The garbage collection behavior for this object is the same as other strongly reachable objects until it is no longer reachable from the root set, at which time it becomes weakly reachable.

Here is another way to look at the same idea:

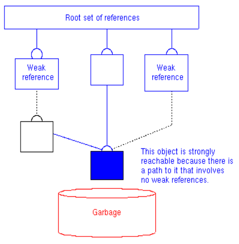
Root set of references
Weak reference
Weak reference
This object is strongly reachable because there is a path to it that involves no weak references.
Garbage

Fig. 3: References suspended from a beam.

Imagine the root set of references as a solid beam from which objects are suspended by cables (strong references). An object might contain references to other objects. Weak reference objects connect to their referent not by a cable, but by a flexible rubber band, no number of which could support an object. Objects not held by cables fall into the garbage bin below.

The Reference Objects API provides several types of reference objects, which gives a program several strengths of object reachability once the referent is no longer strongly reachable. The next section explains the different types of reference objects and what is meant by strengths of reachability.

All About Reference Objects

The Reference Objects API consists of the classes listed below and shown in the figure.

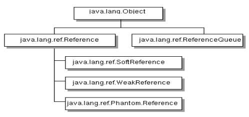
java.lang.Object
java.lang.ref.Reference
java.lang.ref.ReferenceQueue
java.lang.ref.SoftReference
java.lang.ref.WeakReference
java.lang.ref.Phantom.Reference

Fig. 4: Class hierarchy diagram.

The `SoftReference`, `WeakReference`, and `PhantomReference` classes define three kinds of reference objects and three strengths of object reachability. From strongest to weakest, the strengths of reachability are the following:

- Strongly reachable
- Softly rechable

- Weakly reachable
- Phantomly reachable
- Unreachable

Strengths of Reachability

The Reference Objects API defines strengths of object reachability. You can have (going from strongest to weakest) softly, weakly and phantomly reachable objects and gain an amount of interaction with the garbage collector according to the strength of an object's reachability.

There is no limit to the number or type of references there can be to an object. One object can be referenced by any mix of strong, soft, weak, and phantom references depending on your application requirements.

To determine an object's strength of reachability, the garbage collector starts at the root set and traces all the paths to objects on the heap. The reference objects that the garbage collector goes through to reach an object are what determine the reachability of that object. When any path to an object is free of reference objects, the object is strongly reachable. When all paths have one or more reference objects, the object is softly, weakly, or phantomly reachable according to the type of reference object(s) the garbage collector finds. In general, an object is only as reachable as the weakest link in the strongest path from the root set. The sections to follow describe how this all works.

Softly Reachable

An object is *softly reachable* if it is not strongly reachable and there is a path to it with no weak or phantom references, but one or more soft references. The garbage collector might or might not reclaim a softly reachable object depending on how recently the object was created or accessed, but is required to clear all soft references before throwing an `OutOfMemoryError`.

If heap memory is running low, the garbage collector may, at its own discretion, find softly reachable objects that have not been accessed in the longest time and clear them (set their reference field to `null`).

`SoftReference` objects work well in applications that, for example, put a large number of images into memory and there is no way to know if the application (driven by end user input) will access a given image again. If the garbage collector reclaims an image that has not been accessed for a long time, and the application needs to access it again, the application reloads the image and displays it.

Using Soft References in Web-Based Programs

Soft references are very useful in web-based programs where, for example, an applet creates an image to display on a web page. When the user opens the web page, the applet code gets the image and displays it. If the code also creates a soft reference to the image, the garbage collector has the option to reclaim or not reclaim the memory allocated to the image when the user moves to another web page. Here is how the example application looks in memory:

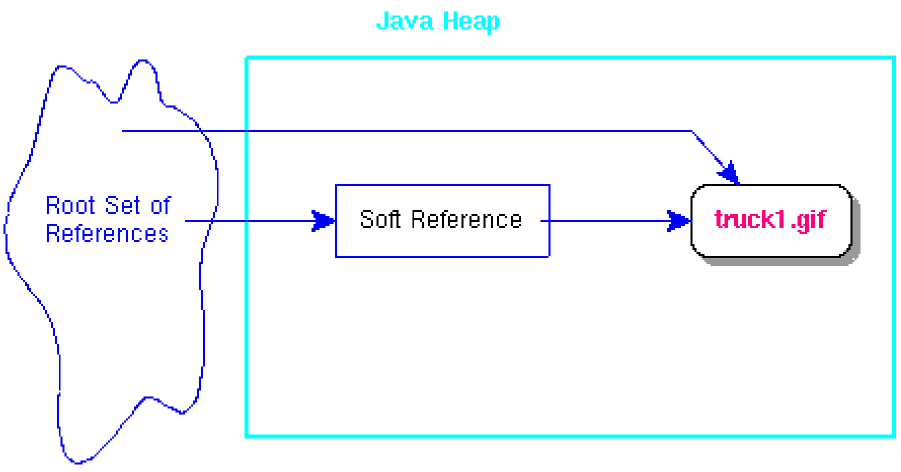

Fig. 5: Using soft references.

If the user returns to the web page where the image is, the applet code uses the SoftReference.get method to check the soft reference to find out if the image is still in memory. If the image has not been reclaimed by the garbage collector, it is quickly displayed on the page; and if the image has been reclaimed, the applet code gets it again.

If your program calls the SoftReference.clear method, the reference may become eligible for reclamation. However, if there is a strong reference to the object, calling the clear method does not cause the garbage collector to reclaim the object. In this case, the soft reference is null, but garbage collection does not happen until the object is no longer strongly reachable.

Here is the complete DisplayImage.java source code.

> **NOTE**: The `im` object is set to `null` because there is no guarantee the stack slot occupied by it will be cleared when it goes out of scope. A later method invocation whose stack frame contains the slot previously occupied

by `im` might not put a new value there, in which case, the garbage collector still considers the slot to contain a root.

This is a problem even with nonconservative exact collectors. As a precaution, you can increase the probability that an object will become softly, weakly, finalizable, or phantomly reachable by clearing variables that refer to it.

```
import java.awt.Graphics;
import java.awt.Image;
import java.applet.Applet;
import java.lang.ref.SoftReference;

public class DisplayImage extends Applet {

        SoftReference sr = null;

        public void init() {
            System.out.println("Initializing");
        }

        public void paint(Graphics g) {
            Image im = (
              sr == null) ? null : (Image)(
              sr.get());
            if (im == null) {
                System.out.println(
                "Fetching image");
                im = getImage(getCodeBase(),
                   "truck1.gif");
                sr = new SoftReference(im);
           }
           System.out.println("Painting");
           g.drawImage(im, 25, 25, this);
           im = null;
        /* Clear the strong reference to the image */
        }

        public void start() {
            System.out.println("Starting");
        }

        public void stop() {
            System.out.println("Stopping");
        }

}
```

Reference Queues

A java.lang.ref.ReferenceQueue is a simple data structure onto which the garbage collector places reference objects when the reference field is cleared (set to `null`). You would use a reference queue to find out when an object becomes softly, weakly, or phantomly reachable so your program can take some action based on that knowledge. For example, a program might perform some post-finalization cleanup processing that requires an object to be unreachable (such as the deallocation of resources outside the Java heap) upon learning that an object has become phantomly reachable.

To be placed on a reference queue, a reference object must be created with a reference queue. Soft and weak reference objects can be created with a reference queue or not, but phantom reference objects must be created with a reference queue:

```
ReferenceQueue queue = new ReferenceQueue();
  PhantomReference pr =
    new PhantomReference(object, queue);
```

Another approach to the soft reference example from the diagram on the previous page could be to create the SoftReference objects with a reference queue, and poll the queue to find out when an image has been reclaimed (its reference field cleared). At that point, the program can remove the corresponding entry from the hash map, and thereby, allow the associated string to be garbage collected.

Note: A program can also wait indefinitely on a reference queue with the `remove()` method, or for a bounded amount of time with the `remove(long timeout)` method.

Soft and weak reference objects are placed in their reference queue some time after they are cleared (their reference field is set to `null`). Phantom reference objects are placed in their reference queue after they become phantomly reachable, but before their reference field is cleared. This is so a program can perform post-finalization cleanup and clear the phantom reference upon completion of the cleanup.

Weakly Reachable

An object is weakly reachable when the garbage collector finds no strong or soft references, but at least one path to the object with a weak reference. Weakly reachable objects are finalized some time after their weak references have been cleared. The only real difference between a soft reference and a weak reference is that the garbage collector uses algorithms to decide whether or not to reclaim a softly reachable object, but always reclaims a weakly reachable object.

Weak references work well in applications that need to, for example, associate extra data with an unchangeable object, such as a thread the application did not create. If you make a weak reference to the thread with a reference queue, your program can be notified when the thread is no longer strongly reachable. Upon receiving this notification, the program can perform any required cleanup of the associated data object.

To make the Thread object weakly reachable, its strong reference must be set to `null`. After the garbage collector clears the weak reference to the thread, it places the weak reference on the queue. When the program removes the reference from the queue, it can then remove the corresponding hash map entry, and thereby, allow the data object to be reclaimed.

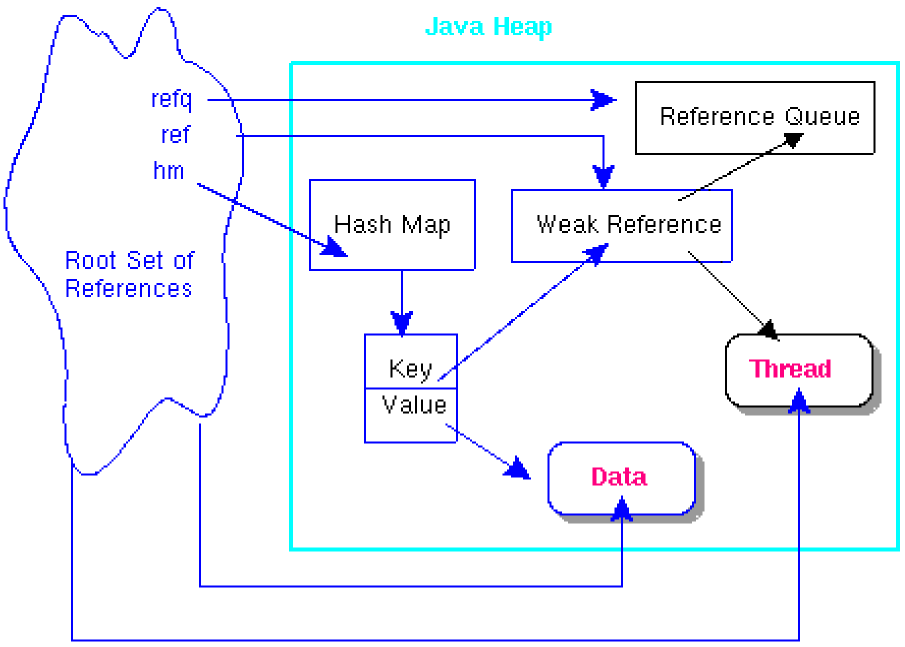

Fig. 6: Using weak references.

The following code examples stores an object and some extra data in a hash map. The object is referenced by a weak reference object that was created with a reference queue. This is a very simple code example to illustrate the points discussed in this article. In a real application, the data would be associated with the object well after the object is created; otherwise, you could keep the data in a subclass of the object instead of using a weak reference.

After the object becomes weakly reachable, the garbage collector clears the weak reference and places it in the reference queue. The code queries the reference queue for the weak reference, and upon finding the weak reference there, sets the reference to the extra data to `null` so the extra data is garbage-collected after the object is.

An important point in this example is that the object can be reached from the stack through two paths: one path goes through the weak reference and the other goes through the hash map. The object does not become weakly reachable until the reference to it in the hash map is null. It is not enough to just set the object to null.

```
import java.lang.ref.*;
import java.util.*;

public class WeakObj {

 public static void main(String[] args) {
  try {
    ReferenceQueue aReferenceQueue
      = new ReferenceQueue();
    Object anObject = new Object();
    WeakReference ref = new WeakReference(anObject,
                             aReferenceQueue);
    String extraData = new String("Extra Data");
    HashMap aHashMap = new HashMap();;
//Associate extraData (value) with weak reference
// (key) in aHashMap
    aHashMap.put(ref, extraData);

//Check that a reference to an object was created
    System.out.println(
      "*** created ref to some object");
    System.out.println();
    System.out.println(
      "contents of ref: " + ref.get());
    System.out.println();


//Check whether the Reference Object is enqueued
    System.out.println(
      "ref.isEnqueued(): " + ref.isEnqueued());
    System.out.println();

//Clear the strong reference to anObject
    anObject = null;

//Clear the strong reference to extraData
    if(anObject == null){
      extraData = null;
    }
//Run the garbage collector, and
//Check the reference object's referent
    System.out.println("*** running gc...");
    System.gc();
    System.out.println();
    System.out.println(
      "contents of ref: " + ref.get());
    System.out.println();


//Check whether the reference object is enqueued
    System.out.println(
      "ref.isEnqueued(): " + ref.isEnqueued());
    System.out.println();

//Enqueue the reference object.
//This method returns false
//if the reference object is already enqueued.
    System.out.println("enqueued="+ref.enqueue());

  } catch (Exception e) {
    System.err.println("An exception occurred:");
    e.printStackTrace();
  }
 }
}
```

Phantomly Reachable

An object is phantomly reachable when the garbage collector finds no strong, soft, or weak references, but at least one path to the object with a phantom reference. Phantomly reachable objects are objects that have been finalized, but not reclaimed.

When the garbage collector finds only phantom references to an object, it enqueues the phantom reference. The program polls the reference queue and upon notification that the phantom reference is enqueued, performs post-finalization cleanup processing that requires the object to be unreachable (such as the deallocation of resources outside the Java heap). At the end of the post-finalization cleanup code, the program should call the `clear()` method on the phantom reference object to set the reference field to `null` to make the referent eligible for garbage collection.

Chains of reference objects

A given path to an object can contain more than one kind of reference object, which creates a chain of reference objects. The garbage collector processes reference objects in order from strongest to weakest. The processing always happens in the following order, but there is no guarantee when the processing will occur:

- Soft references
- Weak references
- Finalization
- Phantom references
- Reclamation

In the diagram below, there is a chain of reference objects to Some Object. When the garbage collector processes the reference objects, will it find Some Object to be phantomly, weakly, or softly reachable?

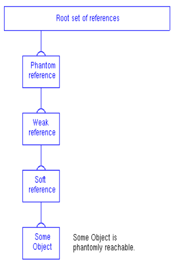

Fig. 7: How reachable is some object?

The answer is Some object is phantomly reachable. In general an object is only as reachable as the weakest link in the strongest path from a root set. In the diagram, the phantom reference is strongly reachable, but all other objects are phantomly reachable.

However, if the weak reference is strongly reachable as shown in the next diagram, both Phantom and Weak reference are strongly reachable, and Soft reference and Some object are weakly reachable.

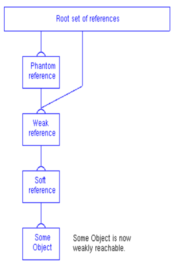

Fig. 8: Phantomly or weakly reachable?

Can you guess what happens if Soft reference is strongly reachable?

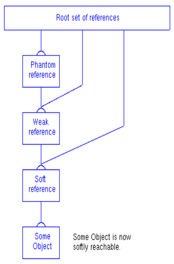

Fig. 9: Phantomly, weakly, or softly reachable?

The answer is Phantom, Weak, and Soft reference are strongly reachable, and Some Object is softly reachable.

WeakHashMap Class

A `WeakHashMap` object is like a `HashMap` object in that it stores key-value pairs where the key is an arbitrary object. But, in a `WeakHashMap` object, the key is referenced by a weak reference object internal to the `WeakHashMap` object.

After the object referenced by the weak key becomes weakly reachable, the garbage collector clears the internal weak reference. At this point, the key and its associated value become eligible for finalization, and if there are no phantom references to the key, the key and value then become eligible for reclamation. The WeakHashMap class provides a weak hash table facility, but in a class that fits into the new Collections framework.

If you use a `WeakHashMap` object instead of the `HashMap` and `WeakReference` objects, you can associate pairs of objects as keys and values. The value is made eligible for garbage collection when the key becomes weakly reachable. This uses fewer lines of code because you do not need a `WeakReference` object, and you do not have to explicitly set the associated object to `null`.