Guidelines for using the Java 2 reference classes

Learn to effectively use SoftReference, WeakReference, and PhantomReference

Peter Haggar

October 01, 2002

The Java 2 platform introduced the `java.lang.ref` package, which contains classes that allow you to refer to objects without pinning them in memory. The classes also provide a limited degree of interaction with the garbage collector. In this article, Peter Haggar examines the functionality and behavior of the `SoftReference`, `WeakReference`, and `PhantomReference` classes and recommends programming idioms for their use.

When the `java.lang.ref` package, which includes the `SoftReference`, `WeakReference`, and `PhantomReference` classes, was first introduced in the Java 2 platform, its usefulness was arguably over-hyped. The classes it contains can be useful, but they have certain limitations that narrow their appeal and make their application very specific to a defined set of problems.

Garbage collection overview

The main feature of the reference classes is the ability to refer to an object that can still be reclaimed by the garbage collector. Before the reference classes were introduced, only strong references were available. For example, the following line of code shows a strong reference, `obj`:

```
Object obj = new Object();
```

The reference `obj` refers to an object stored in the heap. As long as the `obj` reference exists, the garbage collector will never free the storage used to hold the object.

When `obj` goes out of scope or is explicitly assigned to `null`, the object is available for collection, assuming there are no other references to it. However, an important detail to note is that just because an object is available for collection does not mean it will be reclaimed by a given run of the garbage collector. Because garbage collection algorithms vary, some algorithms analyze older, longer-lived objects less frequently than short-lived objects. Thus, an object that is available for collection may never be reclaimed. This can occur if the program ends prior to the garbage collector freeing the object. Therefore, the bottom line is that you can never guarantee that an available object will ever be collected by the garbage collector.

This information is important as you analyze the reference classes. They are useful classes for specific problems, although, due to the nature of garbage collection, they might not prove as useful as you originally thought. Soft, weak, and phantom reference objects offer three different ways to refer to a heap object without preventing its collection. Each type of reference object has different behaviors, and their interaction with the garbage collector varies. In addition, the new reference classes all represent a "weaker" form of a reference than the typical strong reference. Furthermore, an object in memory can be referenced by multiple references that could be strong, soft, weak, or phantom. Before proceeding further, let's look at some terminology:

- **Strongly reachable:** An object that can be accessed by a strong reference.
- **Softly reachable:** An object that is not strongly reachable and can be accessed through a soft reference.
- **Weakly reachable:** An object that is not strongly or softly reachable and can be accessed through a weak reference.
- **Phantomly reachable:** An object that is not strongly, softly, or weakly reachable, has been finalized, and can be accessed through a phantom reference.
- **Clear:** Setting the reference object's referent field to `null` and declaring the object in the heap that the reference class referred to as *finalizable*.

The SoftReference class

A typical use of the `SoftReference` class is for a memory-sensitive cache. The idea of a `SoftReference` is that you hold a reference to an object with the guarantee that all of your soft references will be cleared before the JVM reports an out-of-memory condition. The key point is that when the garbage collector runs, it may or may not free an object that is softly reachable. Whether the object is freed depends on the algorithm of the garbage collector as well as the amount of memory available while the collector is running.

The WeakReference class

A typical use of the `WeakReference` class is for canonicalized mappings. In addition, weak references are useful for objects that would otherwise live for a long time and are also inexpensive to re-create. The key point is that when the garbage collector runs, if it encounters a weakly reachable object, it will free the object the `WeakReference` refers to. Note, however, that it may take multiple runs of the garbage collector before it finds and frees a weakly reachable object.

The PhantomReference class

The `PhantomReference` class is useful only to track the impending collection of the referring object. As such, it can be used to perform pre-mortem cleanup operations. A `PhantomReference` must be used with the `ReferenceQueue` class. The `ReferenceQueue` is required because it serves as the mechanism of notification. When the garbage collector determines an object is phantomly reachable, the `PhantomReference` object is placed on its `ReferenceQueue`. The placing of the `PhantomReference` object on the `ReferenceQueue` is your notification that the object the `PhantomReference` object referred to has been finalized and is ready to be collected. This allows you to take action just prior to the object memory being reclaimed.

Garbage collector and reference interaction

Each time the garbage collector runs, it optionally frees object memory that is no longer strongly reachable. If the garbage collector discovers an object that is softly reachable, the following occurs:

- The `SoftReference` object's referent field is set to `null`, thereby making it not refer to the `heap` object any longer.
- The `heap` object that had been referenced by the `SoftReference` is declared `finalizable`.
- When the `heap` object's `finalize()` method is run and its memory freed, the `SoftReference` object is added to its `ReferenceQueue`, if it exists.

If the garbage collector discovers an object that is weakly reachable, the following occurs:

- The `WeakReference` object's referent field is set to `null`, thereby making it not refer to the `heap` object any longer.
- The `heap` object that had been referenced by the `WeakReference` is declared `finalizable`.
- When the `heap` object's `finalize()` method is run and its memory freed, the `WeakReference` object is added to its `ReferenceQueue`, if it exists.

If the garbage collector discovers an object that is phantomly reachable, the following occurs:

- The `heap` object that is referenced by the `PhantomReference` is declared `finalizable`.
- Unlike soft and weak references, the `PhantomReference` is added to its `ReferenceQueue` before the heap object is freed. (Remember, all `PhantomReference` objects must be created with an associated `ReferenceQueue`.) This allows for action to be taken before the heap object is reclaimed.

Consider the code in Listing 1. Figure 1 illustrates the execution of this code.

```
//Create a strong reference to an object
MyObject obj = new MyObject();                  //1

//Create a reference queue
ReferenceQueue rq = new ReferenceQueue();       //2

//Create a weakReference to obj and associate our reference queue
WeakReference wr = new WeakReference(obj, rq);  //3
```

Figure 1. Object layout after executing lines //1, //2, and //3 of code in Listing 1

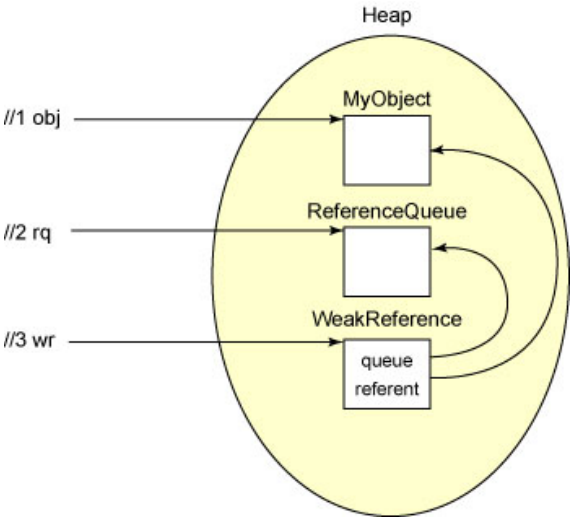


Figure 1 shows the state of the objects after each line of code executes. Line //1 creates the object `MyObject`, while line //2 creates the `ReferenceQueue` object. Line //3 creates the `WeakReference` object that refers to its referent, `MyObject`, and its `ReferenceQueue`. Note that each object reference, `obj`, `rq`, and `wr`, are all strong references. To take advantage of these reference classes, you must break the strong reference to the `MyObject` object by setting `obj` to `null`. Recall that if you do not do this, object `MyObject` will never be reclaimed, mitigating any benefit of the reference classes.

Each of the reference classes has a `get()` method, and the `ReferenceQueue` class has a `poll()` method. The `get()` method returns the reference to the referent object. Calling `get()` on a `PhantomReference` always returns `null`. This is because `PhantomReference`s are only used to track collection. The `poll()` method returns the reference object that has been added to the queue, and `null` if nothing is on the queue. Therefore, the result of calling `get()` and `poll()` after the execution of Listing 1 would be:

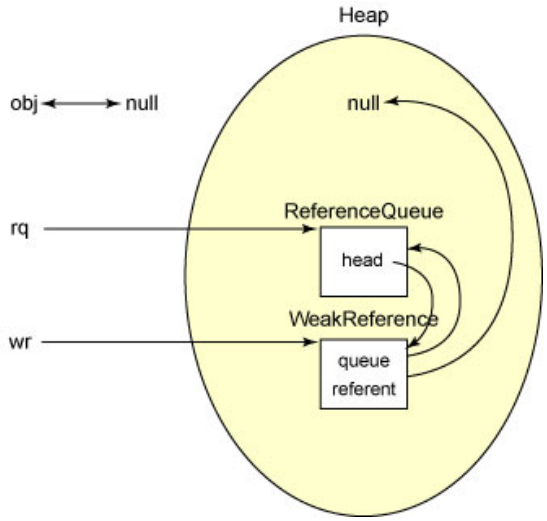
```
wr.get();   //returns reference to MyObject
rq.poll();  //returns null
```

Now, assume the garbage collector runs. The `get()` and `poll()` methods return the same values because the `MyObject` object is not freed; `obj` still maintains a strong reference to it. In fact, the object layout remains unchanged and looks like Figure 1. However, consider this code:

```
obj = null;
System.gc();  //run the collector
```

After this code executes, the object layout is as depicted in Figure 2:

Figure 2. Object layout after obj = null; and garbage collector is run



Now, calling `get()` and `poll()` yields different results:

```
wr.get();   //returns null
rq.poll();  //returns a reference to the WeakReference object
```

This situation indicates that the `MyObject` object, whose reference was originally held by the `WeakReference` object, is no longer available. This means that the garbage collector freed the memory for `MyObject`, enabling the `WeakReference` object to be placed on its `ReferenceQueue`. Therefore, you know that an object has been declared `finalizable` and possibly, but not necessarily, collected, when the `get()` method of the `WeakReference` or `SoftReference` class returns `null`. Only when finalization is completed and the `heap` object's memory is collected is the `WeakReference` or `SoftReference` placed on its associated `ReferenceQueue`. Listing 2 shows a full working program demonstrating some of these principles. The code is relatively self-explanatory with many comments and print statements to aid in its understanding.

```
import java.lang.ref.*;
class MyObject
{
  protected void finalize() throws Throwable
  {
    System.out.println("In finalize method for this object: " +
                       this);
  }
}

class ReferenceUsage
{
  public static void main(String args[])
  {
    hold();
    release();
  }

  public static void hold()
  {
    System.out.println("Example of incorrectly holding a strong " +
                       "reference");
    //Create an object
```

```
    MyObject obj = new MyObject();
    System.out.println("object is " + obj);

    //Create a reference queue
    ReferenceQueue rq = new ReferenceQueue();

    //Create a weakReference to obj and associate our reference queue
    WeakReference wr = new WeakReference(obj, rq);

    System.out.println("The weak reference is " + wr);

    //Check to see if it's on the ref queue yet
    System.out.println("Polling the reference queue returns " +
                       rq.poll());
    System.out.println("Getting the referent from the " +
                       "weak reference returns " + wr.get());

    System.out.println("Calling GC");
    System.gc();
    System.out.println("Polling the reference queue returns " +
                       rq.poll());
    System.out.println("Getting the referent from the " +
                       "weak reference returns " + wr.get());
  }

  public static void release()
  {
    System.out.println("");
    System.out.println("Example of correctly releasing a strong " +
                       "reference");
    //Create an object
    MyObject obj = new MyObject();
    System.out.println("object is " + obj);

    //Create a reference queue
    ReferenceQueue rq = new ReferenceQueue();

    //Create a weakReference to obj and associate our reference queue
    WeakReference wr = new WeakReference(obj, rq);

    System.out.println("The weak reference is " + wr);

    //Check to see if it's on the ref queue yet
    System.out.println("Polling the reference queue returns " +
                       rq.poll());
    System.out.println("Getting the referent from the " +
                       "weak reference returns " + wr.get());

    System.out.println("Set the obj reference to null and call GC");
    obj = null;
    System.gc();
    System.out.println("Polling the reference queue returns " +
                       rq.poll());
    System.out.println("Getting the referent from the " +
                       "weak reference returns " + wr.get());
  }
}
```

Usage and idioms

The idea behind these classes is to avoid pinning an object in memory for the duration of the application. Instead, you softly, weakly, or phantomly refer to an object, allowing the garbage collector to optionally free it. This usage can be beneficial when you want to minimize the amount of heap memory an application uses over its lifetime. You must remember that to make use of

these classes you cannot maintain a strong reference to the object. If you do, you waste any benefit the classes offer.

In addition, you must use the correct programming idiom to check if the collector has reclaimed the object before using it, and if it has, you must re-create the object first. This process can be done with different programming idioms. Choosing the wrong idiom can cause you problems. Consider the code idiom in Listing 3 for retrieving the referent object from a `WeakReference`:

```
obj = wr.get();
if (obj == null)
{
  wr = new WeakReference(recreateIt());  //1
  obj = wr.get();                        //2
}
//code that works with obj
```

After studying this code, consider the alternate code idiom in Listing 4 for retrieving the referent object from a `WeakReference`:

```
obj = wr.get();
if (obj == null)
{
  obj = recreateIt();                    //1
  wr = new WeakReference(obj);           //2
}
//code that works with obj
```

Compare the two idioms to see if you can determine which one is guaranteed to work, and which one is not. The idiom outlined in Listing 3 is not guaranteed to work at all times, but Listing 4's idiom is. The reason the idiom in Listing 3 is deficient is because `obj` is not guaranteed to be non-null after the body of the `if` block completes. Consider what happens if the garbage collector runs after line //1 of Listing 3 but before line //2 executes. The `recreateIt()` method re-creates the object, but it is referenced by a `WeakReference`, not a strong reference. Therefore, if the collector runs before line //2 attaches a strong reference to the re-created object, the object is lost and `wr.get()` returns `null`.

Listing 4 does not have this problem because line //1 re-creates the object and assigns a strong reference to it. Therefore, if the garbage collector runs after this line, but before line //2, the object will not be reclaimed. Then, the `WeakReference` to `obj` is created at line //2. After working with `obj` after this `if` block, you should set `obj` to `null` to enable the garbage collector to reclaim this object to take full advantage of weak references. Listing 5 shows a complete program demonstrating the idiom differences just described. (To run this program, a "temp.fil" file must exist in the directory from which the program runs.)

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

class ReferenceIdiom
{
  public static void main(String args[]) throws FileNotFoundException
  {
    broken();
    correct();
```

```
  }

  public static FileReader recreateIt() throws FileNotFoundException
  {
    return new FileReader("temp.fil");
  }

  public static void broken() throws FileNotFoundException
  {
    System.out.println("Executing method broken");
    FileReader obj = recreateIt();
    WeakReference wr = new WeakReference(obj);

    System.out.println("wr refers to object " + wr.get());

    System.out.println("Now, clear the reference and run GC");
    //Clear the strong reference, then run GC to collect obj.
    obj = null;
    System.gc();

    System.out.println("wr refers to object " + wr.get());

    //Now see if obj was collected and recreate it if it was.
    obj = (FileReader)wr.get();
    if (obj == null)
    {
      System.out.println("Now, recreate the object and wrap it
        in a WeakReference");
      wr = new WeakReference(recreateIt());
      System.gc();  //FileReader object is NOT pinned...there is no
                    //strong reference to it.  Therefore, the next
                    //line can return null.
      obj = (FileReader)wr.get();
    }
    System.out.println("wr refers to object " + wr.get());
  }

  public static void correct() throws FileNotFoundException
  {
    System.out.println("");
    System.out.println("Executing method correct");
    FileReader obj = recreateIt();
    WeakReference wr = new WeakReference(obj);

    System.out.println("wr refers to object " + wr.get());

    System.out.println("Now, clear the reference and run GC");
    //Clear the strong reference, then run GC to collect obj
    obj = null;
    System.gc();

    System.out.println("wr refers to object " + wr.get());

    //Now see if obj was collected and recreate it if it was.
    obj = (FileReader)wr.get();
    if (obj == null)
    {
      System.out.println("Now, recreate the object and wrap it
        in a WeakReference");
      obj = recreateIt();
      System.gc();  //FileReader is pinned, this will not affect
                    //anything.
      wr = new WeakReference(obj);
    }
    System.out.println("wr refers to object " + wr.get());
  }
}
```

Summary

The Reference classes can be useful if used for the right situations. However, their usefulness is tempered by the fact that they rely on the sometimes unpredictable behavior of the garbage collector. Their effective use is also dependent on applying the correct programming idioms; it's critical that you understand how these classes are implemented and how to program to them.

Related topic

- Jeff Friesen demonstrates how to use the Reference Object API to manage image caches, obtain notification when significant objects are no longer strongly reachable, and perform post-finalization cleanup in this article from *JavaWorld* (January 2002).