1. General information

https://stackoverflow.com/questions/8790511/is-there-a-practical-use-for-weak-references

The most common usage of weak references is for values in "lookup" Maps. This can be split futrher as:

1. For holding additional (often expensively calculated but reproducible) information about specific objects that you cannot modify directly, and whose lifecycle you have little control over. WeakHashMap is a perfect way of holding these references: the key in the WeakHashMap is only weakly held, and so when the key is garbage collected, the value can be removed from the Map too, and hence be garbage collected.
2. For implementing some kind of eventing or notification system, where "listeners" are registered with some kind of coordinator, so they can be informed when something occurs – but where you don't want to prevent these listeners from being garbage collected when they come to the end of their life. A WeakReference will point to the object while it is still alive, but point to "null" once the original object has been garbage collected.

2. Understanding Weak References

Posted by **enicholas** on May 4, 2006 at 5:06 PM PDT

Some time ago I was interviewing candidates for a Senior Java Engineer position. Among the many questions I asked was "What can you tell me about weak references?" I wasn't expecting a detailed technical treatise on the subject. I would probably have been satisfied with "Umm... don't they have something to do with garbage collection?" I was instead surprised to find that out of twenty-odd engineers, all of whom had at least five years of Java experience and good qualifications, only *two* of them even knew that weak references existed, and only one of those two had actual useful knowledge about them. I even explained a bit about them, to see if I got an "Oh yeah" from anybody -- nope. I'm not sure why this knowledge is (evidently) uncommon, as weak references are a massively useful feature which have been around since Java 1.2 was released, over seven years ago.

Now, I'm not suggesting you need to be a weak reference expert to qualify as a decent Java engineer. But I humbly submit that you should at least *know what they are* -- otherwise how will you know when you should be using them? Since they seem to be a little-known feature, here is a brief overview of what weak references are, how to use them, and when to use them.

Strong references

First I need to start with a refresher on *strong references*. A strong reference is an ordinary Java reference, the kind you use every day. For example, the code:

```
StringBuffer buffer = new StringBuffer();
```

creates a new `StringBuffer()` and stores a strong reference to it in the variable `buffer`. Yes, yes, this is kiddie stuff, but bear with me. The important part about strong references -- the part that makes them "strong" -- is how they interact with the garbage collector. Specifically, if an object is reachable via a chain of strong references (strongly reachable), it is not eligible for garbage collection. As you don't want the garbage collector destroying objects you're working on, this is normally exactly what you want.

When strong references are too strong

It's not uncommon for an application to use classes that it can't reasonably extend. The class might simply be marked `final`, or it could be something more complicated, such as an interface returned by a factory method backed by an unknown (and possibly even unknowable) number of concrete implementations. Suppose you have to use a class `Widget` and, for whatever reason, it isn't possible or practical to extend `Widget` to add new functionality.

What happens when you need to keep track of extra information about the object? In this case, suppose we find ourselves needing to keep track of each `Widget's` serial number, but the `Widget` class doesn't actually have a serial number property -- and because `Widget` isn't extensible, we can't add one. No problem at all, that's what `HashMaps` are for:

```
serialNumberMap.put(widget, widgetSerialNumber);
```

This might look okay on the surface, but the strong reference to `widget` will almost certainly cause problems. We have to know (with 100% certainty) when a particular `Widget's` serial number is no longer needed, so we can remove its entry from the map. Otherwise we're going to have a memory leak (if we don't remove `Widgets` when we should) or we're going to inexplicably find ourselves missing serial numbers (if we remove `Widgets` that we're still using). If these problems sound familiar, they should: they are exactly the problems that users of non-garbage-collected languages face when trying to manage memory, and we're not supposed to have to worry about this in a more civilized language like Java.

Another common problem with strong references is caching, particular with very large structures like images. Suppose you have an application which has to work with user-supplied images, like the web site design tool I work on. Naturally you want to cache these images, because loading them from disk is very expensive and you want to avoid the possibility of having two copies of the (potentially gigantic) image in memory at once.

Because an image cache is supposed to prevent us from reloading images when we don't absolutely need to, you will quickly realize that the cache should always contain a reference to any image which is already in memory. With ordinary strong references, though, that reference itself will force the image to remain in memory, which requires you (just as above) to somehow determine when the image is no longer needed in memory and remove it from the cache, so that it becomes eligible for garbage collection. Once again you are forced to duplicate the behavior of the garbage collector and manually determine whether or not an object should be in memory.

Weak references

A *weak reference*, simply put, is a reference that isn't strong enough to force an object to remain in memory. Weak references allow you to leverage the garbage collector's ability to determine reachability for you, so you don't have to do it yourself. You create a weak reference like this:

```
WeakReference<Widget> weakWidget = new WeakReference<Widget>(widget);
```

and then elsewhere in the code you can use `weakWidget.get()` to get the actual `Widget` object. Of course the weak reference isn't strong enough to prevent garbage collection, so you may find (if there are no strong references to the widget) that `weakWidget.get()` suddenly starts returning `null`.

To solve the "widget serial number" problem above, the easiest thing to do is use the built-in `WeakHashMap` class. `WeakHashMap` works exactly like `HashMap`, except that the keys (*not* the values!) are referred to using weak references. If a `WeakHashMap` key becomes garbage, its entry is removed automatically. This avoids the pitfalls I described and requires no changes other than the switch from `HashMap` to a `WeakHashMap`. If you're following the standard convention of referring to your maps via the `Map` interface, no other code needs to even be aware of the change.

Reference queues

Once a `WeakReference` starts returning `null`, the object it pointed to has become garbage and the `WeakReference` object is pretty much useless. This generally means that some sort of cleanup is required; `WeakHashMap`, for example, has to remove such defunct entries to avoid holding onto an ever-increasing number of dead `WeakReferences`.

The `ReferenceQueue` class makes it easy to keep track of dead references. If you pass a `ReferenceQueue` into a weak reference's constructor, the reference object will be automatically inserted into the reference queue when the object to which it pointed becomes garbage. You can then, at some regular interval, process the `ReferenceQueue` and perform whatever cleanup is needed for dead references.

Different degrees of weakness

Up to this point I've just been referring to "weak references", but there are actually four different degrees of reference strength: strong, soft, weak, and phantom, in order from strongest to weakest. We've already discussed strong and weak references, so let's take a look at the other two.

Soft references

A *soft reference* is exactly like a weak reference, except that it is less eager to throw away the object to which it refers. An object which is only weakly reachable (the strongest references to it are `WeakReferences`) will be discarded at the next garbage collection cycle, but an object which is softly reachable will generally stick around for a while.

`SoftReferences` aren't *required* to behave any differently than `WeakReferences`, but in practice softly reachable objects are generally retained as long as memory is in plentiful supply. This makes them an excellent foundation for a cache, such as the image cache described above, since you can let the garbage collector worry about both how reachable the objects are (a strongly reachable object will *never* be removed from the cache) and how badly it needs the memory they are consuming.

Phantom references

A *phantom reference* is quite different than either `SoftReference` or `WeakReference`. Its grip on its object is so tenuous that you can't even retrieve the object -- its `get()` method always returns `null`. The only use for such a reference is keeping track of when it gets enqueued into a `ReferenceQueue`, as at that point you know the object to which it pointed is dead. How is that different from `WeakReference`, though?

The difference is in exactly when the enqueuing happens. `WeakReferences` are enqueued as soon as the object to which they point becomes weakly reachable. This is *before* finalization or garbage collection has actually happened; in theory the object could even be "resurrected" by an unorthodox `finalize()` method, but the `WeakReference` would remain dead. `PhantomReferences` are enqueued only when the object is physically removed from memory, and the `get()` method always returns `null` specifically to prevent you from being able to "resurrect" an almost-dead object.

What good are `PhantomReferences`? I'm only aware of two serious cases for them: first, they allow you to determine exactly when an object was removed from memory. They are in fact the *only* way to determine that. This isn't generally that useful, but might come in handy in certain very specific circumstances like manipulating large images: if you know for sure that an image should be garbage collected, you can wait until it actually is before attempting to load the next image, and therefore make the dreaded `OutOfMemoryError` less likely.

Second, `PhantomReferences` avoid a fundamental problem with finalization: `finalize()` methods can "resurrect" objects by creating new strong references to them. So what, you say? Well, the problem is that an object which overrides `finalize()` must now be determined to be garbage in at least two separate garbage collection cycles in order to be collected. When the first cycle determines that it is garbage, it becomes eligible for finalization. Because of the (slim, but unfortunately real) possibility that the object was "resurrected" during finalization, the garbage collector has to run again before the object can actually be removed. And because finalization might not have happened in a timely fashion, an arbitrary number of garbage collection cycles might have happened while the object was waiting for finalization. This can mean serious delays in actually cleaning up garbage objects, and is why you can get `OutOfMemoryErrors` even when most of the heap is garbage.

With `PhantomReference`, this situation is impossible -- when a `PhantomReference` is enqueued, there is absolutely no way to get a pointer to the now-dead object (which is good, because it isn't in memory any longer). Because `PhantomReference` cannot be used to resurrect an object, the object can be instantly cleaned up during the first garbage collection cycle in which it is found to be phantomly reachable. You can then dispose whatever resources you need to at your convenience.

Arguably, the `finalize()` method should never have been provided in the first place. `PhantomReferences` are definitely safer and more efficient to use, and eliminating `finalize()` would have made parts of the VM considerably simpler. But, they're also more work to implement, so I confess to still using `finalize()` most of the time. The good news is that at least you have a choice.

Conclusion

I'm sure some of you are grumbling by now, as I'm talking about an API which is nearly a decade old and haven't said anything which hasn't been said before. While that's certainly true, in my experience many Java programmers really don't know very much (if anything) about weak references, and I felt that a refresher course was needed. Hopefully you at least learned a *little* something from this review.

3. Another tutorial

This Tech Tip reprinted with permission by java.sun.comThe May 11, 1999 Tech Tip titled Reference Objects

introduced the concept of reference objects, but didn't go into much depth. In this tip, you'll learn more about this topic. Basically, reference objects provide a way to indirectly reference the memory needed by objects. The reference objects are maintained in a reference queue (class `ReferenceQueue`), which monitors the reference objects for reachability. Based on the type of reference object, the garbage collector can free memory at times when a normal object reference might not be released.

In the Java platform, there are four types of references to objects. Direct references are the type you normally use, as in:

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

You can think of direct references as strong references that require no extra coding to create or access the object. The remaining three types of references are subclasses of the

`Reference` class found in the `java.lang.ref` package. Soft references are provided by the `SoftReference` class, weak references by the `WeakReference` class, and phantom references by `PhantomReference`.

Soft references act like a data cache. When system memory is low, the garbage collector can arbitrarily free an object whose only reference is a soft reference. In other words, if there are no strong references to an object, that object is a candidate for release. The garbage collector is required to release any soft references before throwing an `OutOfMemoryException`.

Weak references are weaker than soft references. If the only references to an object are weak references, the garbage collector can reclaim the memory used by an object at any time. There is no requirement for a low memory situation. Typically, memory used by the object is reclaimed in the next pass of the garbage collector.

Phantom references relate to cleanup tasks. They offer a notification immediately before the garbage collector performs the finalization process and frees an object. Consider it a way to do cleanup tasks within an object.

As a simple demonstration of using a `WeakHashMap`, the following `WeakTest` program creates a `WeakHashMap` with a single element in it. It then creates a second thread that waits for the map to empty out, requesting that the garbage collector run every 1/2 second. The main thread waits for this second thread to finish.

```
import java.util.*;

public class WeakTest {
  private static Map<String, String> map;
  public static void main (String args[]) {
    map = new WeakHashMap<String, String>();
    map.put(new String("Scott"), "McNealey");
    Runnable runner = new Runnable() {
      public void run() {
        while (map.containsKey("Scott")) {
          try {
            Thread.sleep(500);
          } catch (InterruptedException ignored) {
          }
          System.out.println("Checking for empty");
```

```
              System.gc();
            }
          }
        };

        Thread t = new Thread(runner);
        t.start();
        System.out.println("Main joining");
        try {
          t.join();
        } catch (InterruptedException ignored) {
        }
      }
    }
```

Since there are no strong references to the only key in the map, the entry in the map should be garbage collected at the earliest convenience of the system.

Running the program generates the following results:

```
 Main joining
 Checking for empty
```

There are two important things to point out in this example. First, normally a call to `System.gc()` would not be required. Because the `WeakTest` program is lightweight and doesn't use much memory an explicit call to the garbage collector is necessary. Second, notice the call to `new String("Scott")`. You might ask why call `new String()` when the end result is the same `"Scott"` string? The answer is that if you don't call `new String()`, the reference for the map key will be to the system's string constant pool. This never goes away, so the weak reference will never be released. To get around this, `new String("Scott")` creates a reference to the reference in the string constant pool. The string contents are never duplicated. They stay in the constant pool. This simply creates a separate pointer to the string constant in the pool.

A more complicated use of `WeakHashMap` is to manage a listener list for a Swing component or data model. This would not be a general purpose listener list. Instead it would be a weak listener list. In this case, as long as a strong reference is kept outside the component or data model, that object will continue to notify the listener/observer. This helps garbage collection in the sense that the listener doesn't prevent the source object (which registered the listener) from being garbage collected. By default, listener references are strong references and are not garbage collected when your method returns. You must remember to remove the listener when you finish monitoring the situation.

To demonstrate, lets create a `WeakListModel` that offers a `ListModel`. The `ListModel` relies on a `WeakHashMap` for storing `ListDataListener` objects.

Aside from the imports, the start of the class definition comprises the class and local variable declarations.

```
   public class WeakListModel implements ListModel,
     Serializable {

     private Map<ListDataListener, Object> listenerList =
       Collections.synchronizedMap(
         new WeakHashMap<ListDataListener, Object>());
     private final Object present = new Object();
     private ArrayList<Object> delegate = new ArrayList<Object>();
```

The listener list is a `WeakHashMap`. Its access is synchronized. Although some implementations of listener lists rely on making copies to avoid synchronized access, weak references can be removed at any time. So making a copy provides two places for the weak reference to be dropped with no added value.

Because the listener list is maintained in a `Map`, you need both a key and a value. The value associated with every key in the map is the '`present`' object. Theoretically, you only need a `WeakHashSet`. However, the class is not in the predefined set of collections, so a `WeakHashMap` is used instead.

The last variable, delegate, manages the ListModel contents. Most of the ListModel interface implementation simply passes the request to the delegate to perform the operation, hence the name delegate.

The first set of methods for the WeakListModel are related to sizing or querying the model structure. In all cases, these pass the call to the delegate:

```
public int getSize() {
  return delegate.size();
}

public Object getElementAt(int index) {
  return delegate.get(index);
}

public void trimToSize() {
  delegate.trimToSize();
}

public void ensureCapacity(int minCapacity) {
  delegate.ensureCapacity(minCapacity);
}

public int size() {
  return delegate.size();
}

public boolean isEmpty() {
  return delegate.isEmpty();
}

public Enumeration elements() {
  return Collections.enumeration(delegate);
}

public boolean contains(Object elem) {
  return delegate.contains(elem);
}

public int indexOf(Object elem) {
  return delegate.indexOf(elem);
}

public int lastIndexOf(Object elem) {
  return delegate.lastIndexOf(elem);
}

public Object elementAt(int index) {
  return delegate.get(index);
}

public Object firstElement() {
  return delegate.get(0);
}

public Object lastElement() {
  return delegate.get(delegate.size()-1);
}

public String toString() {
  return delegate.toString();
  }
```

The next set of methods have to do with adding and removing elements. In addition to accessing the `delegate` for the storage operation, the set of listeners need to be notified. These methods call `fire`*XXX*`()` methods that will be shown shortly. These methods do the bulk of the work in the class.

```
  public void setElementAt(Object obj, int index) {
    delegate.set(index, obj);
    fireContentsChanged(this, index, index);
  }

  public void removeElementAt(int index) {
    delegate.remove(index);
    fireIntervalRemoved(this, index, index);
  }

  public void insertElementAt(Object obj, int index) {
    delegate.add(index, obj);
    fireIntervalAdded(this, index, index);
 }

  public void addElement(Object obj) {
    int index = delegate.size();
    delegate.add(obj);
    fireIntervalAdded(this, index, index);
  }

  public boolean removeElement(Object obj) {
    int index = indexOf(obj);
    boolean rv = delegate.remove(obj);
    if (index >= 0) {
      fireIntervalRemoved(this, index, index);
    }
    return rv;
  }

  public void removeAllElements() {
    int index1 = delegate.size()-1;
    delegate.clear();
    if (index1 >= 0) {
      fireIntervalRemoved(this, 0, index1);
    }
  }
```

The listener list itself deals with the add and remove listener calls. Again, the `present` object is not used itself. A "map" just needs a key and value. This is the same pattern used by `TreeSet`, which is backed by a `TreeMap`.

```
  public synchronized void addListDataListener(
   ListDataListener l) {
    listenerList.put(l, present);
  }

  public synchronized void removeListDataListener(
   ListDataListener l) {
    listenerList.remove(l);
  }

  public EventListener[] getListeners(Class listenerType) {
    Set<ListDataListener> set = listenerList.keySet();
    return set.toArray(new EventListener[0]);
  }
```

The most interesting set of methods are the `fire`*XXX*`()` methods. For all three of these methods, `fireContentsChanged()`, `fireIntervalAdded()`, and `fireIntervalRemoved()`, a new `Set` is created with the keys from the `WeakHashMap`. This is done to ensure that if the set changes in the middle of notification, the original set is still notified.

```
  protected synchronized void fireContentsChanged(
      Object source, int index0, int index1) {
    ListDataEvent e = null;

    Set<ListDataListener> set =
      new HashSet<ListDataListener>(listenerList.keySet());
    Iterator<ListDataListener> iter = set.iterator();

    while (iter.hasNext()) {
      if (e == null) {
        e = new ListDataEvent(
         source, ListDataEvent.CONTENTS_CHANGED,
         index0, index1);
      }
      ListDataListener ldl = iter.next();
      ldl.contentsChanged(e);
    }
  }

  protected synchronized void fireIntervalAdded(
      Object source, int index0, int index1) {
    ListDataEvent e = null;

    Set<ListDataListener> set =
      new HashSet<ListDataListener>(listenerList.keySet());
    Iterator<ListDataListener> iter = set.iterator();

    while (iter.hasNext()) {
      if (e == null) {
        e = new ListDataEvent(
         source, ListDataEvent.INTERVAL_ADDED,
         index0, index1);
      }
      ListDataListener ldl = iter.next();
      ldl.intervalAdded(e);
    }
  }

  protected synchronized void fireIntervalRemoved(
      Object source, int index0, int index1) {
    ListDataEvent e = null;

    Set<ListDataListener> set =
      new HashSet<ListDataListener>(listenerList.keySet());

    Iterator<ListDataListener> iter = set.iterator();

    while (iter.hasNext()) {
      if (e == null) {
        e = new ListDataEvent(
         source, ListDataEvent.INTERVAL_REMOVED,
          index0, index1);
      }
      ListDataListener ldl = iter.next();
      ldl.intervalRemoved(e);
    }
  }
```

Add the imports and close the braces and you have your class definition:

```
  import java.io.Serializable;
  import java.util.*;
  import java.lang.ref.*;
  import javax.swing.*;
  import javax.swing.event.*;

  ...

  }
```

To test the `ListModel`, the following program, `TestListModel`, defines a `ListDataListener` and associates it with the created `WeakListModel`. Notice that adding and removing the names of Sun's founders as elements to the model notifies the listener. As soon as the strong reference to the listener is gone, the `WeakListModel` can release the weak listener reference. Further operations on the model then do not notify the listener.

```
  import javax.swing.*;
  import javax.swing.event.*;

  public class TestListModel {
    public static void main (String args[]) {
      ListDataListener ldl = new ListDataListener() {
        public void intervalAdded(ListDataEvent e) {
          System.out.println("Added: " + e);
        }
        public void intervalRemoved(ListDataEvent e) {
          System.out.println("Removed: " + e);
        }
        public void contentsChanged(ListDataEvent e) {
          System.out.println("Changed: " + e);
        }
      };
      WeakListModel model = new WeakListModel();
      model.addListDataListener(ldl);
      model.addElement("Scott McNealy");
      model.addElement("Bill Joy");
      model.addElement("Andy Bechtolsheim");
      model.addElement("Vinod Khosla");
      model.removeElement("Scott McNealy");
      ldl = null;
      System.gc();
      model.addElement("Scott McNealy");
      System.out.println(model);
    }
  }
```

Compile and run the `TestListModel` program. It should generate the following results:

```
> java TestListModel

Added: javax.swing.event.ListDataEvent[type=1,index0=0,index1
=0]
Added: javax.swing.event.ListDataEvent[type=1,index0=1,index1
=1]
Added: javax.swing.event.ListDataEvent[type=1,index0=2,index1
=2]
Added: javax.swing.event.ListDataEvent[type=1,index0=3,index1
=3]
Removed: javax.swing.event.ListDataEvent[type=2,index0=0,inde
x1=0]
 [Bill Joy, Andy Bechtolsheim, Vinod Khosla, Scott McNealy]
```

See the `java.lang.ref` package documentation for additional information about reference objects, notification, and reachability.

4. Examples

```
import java.util.Map;
import java.util.WeakHashMap;

public class MainClass {
  private static Map map;

  public static void main(String args[]) {
    map = new WeakHashMap();
    map.put("A", "B");
    Runnable runner = new Runnable() {
      public void run() {
        while (map.containsKey("A")) {
          try {
            Thread.sleep(1000);
            System.gc();
          } catch (InterruptedException ignored) {
          }
          System.out.println("Has A");
          System.gc();
        }
      }
    };
    Thread t = new Thread(runner);
    t.start();
    System.out.println("Main waiting");
    try {
      t.join();
    } catch (InterruptedException ignored) {
    }
    System.gc();
  }
}
```

To enable automatically release of the value, the value must be wrapped in a WeakReference object

```
import java.lang.ref.WeakReference;
import java.util.Iterator;
import java.util.Map;
```

```java
import java.util.WeakHashMap;

public class Main {
  public static void main(String[] argv) throws Exception {
    Object keyObject = "";
    Object valueObject = "";
    Map weakMap = new WeakHashMap();

    weakMap.put(keyObject, valueObject);
    WeakReference weakValue = new WeakReference(valueObject);

    weakMap.put(keyObject, weakValue);

    Iterator it = weakMap.keySet().iterator();
    while (it.hasNext()) {
      Object key = it.next();
      weakValue = (WeakReference) weakMap.get(key);
      if (weakValue == null) {
        System.out.println("Value has been garbage-collected");
      } else {
        System.out.println("Get value");
        valueObject = weakValue.get();
      }
    }
  }
}
```

---

https://stackoverflow.com/questions/14450538/using-javas-referencequeue

```java
ReferenceQueue<Foo> fooQueue = new ReferenceQueue<Foo>();

class ReferenceWithCleanup extends WeakReference<Foo> {
  Bar bar;
  ReferenceWithCleanup(Foo foo, Bar bar) {
    super(foo, fooQueue);
    this.bar = bar;
  }
  public void cleanUp() {
    bar.cleanUp();
  }
}

public Thread cleanupThread = new Thread() {
  public void run() {
    while(true) {
      ReferenceWithCleanup ref = (ReferenceWithCleanup)fooQueue.remove();
      ref.cleanUp();
    }
  }
}

public void doStuff() {
  cleanupThread.start();
  Foo foo = new Foo();
  Bar bar = new Bar();
  ReferenceWithCleanup ref = new ReferenceWithCleanup(foo, bar);
  ... // From now on, once you release all non-weak references to foo,
      // then at some indeterminate point in the future, bar.cleanUp() will
```

```
        // be run. You can force it by calling ref.enqueue().
}
```

---

https://stackoverflow.com/questions/299659/whats-the-difference-between-softreference-and-weakreference-in-java/299702#299702

Principle: `weak reference` is related to garbage collection. Normally, object having one or more `reference` will not be eligible for garbage collection.
The above principle is not applicable when it is `weak reference`. If an object has only weak reference with other objects, then its ready for garbage collection.

Let's look at the below example: We have an `Map` with Objects where Key is reference a object.

```
import java.util.HashMap;
public class Test {

    public static void main(String args[]) {
        HashMap<Employee, EmployeeVal> aMap = new
                       HashMap<Employee, EmployeeVal>();

        Employee emp = new Employee("Vinoth");
        EmployeeVal val = new EmployeeVal("Programmer");

        aMap.put(emp, val);

        emp = null;

        System.gc();
        System.out.println("Size of Map" + aMap.size());

    }
}
```

Now, during the execution of the program we have made `emp = null`. The `Map` holding the key makes no sense here as it is `null`. In the above situation, the object is not garbage collected.

WeakHashMap

`WeakHashMap` is one where the entries (`key-to-value mappings`) will be removed when it is no longer possible to retrieve them from the `Map`.

Let me show the above example same with **WeakHashMap**

```
import java.util.WeakHashMap;

public class Test {

    public static void main(String args[]) {
        WeakHashMap<Employee, EmployeeVal> aMap =
                    new WeakHashMap<Employee, EmployeeVal>();

        Employee emp = new Employee("Vinoth");
        EmployeeVal val = new EmployeeVal("Programmer");

        aMap.put(emp, val);

        emp = null;

        System.gc();
        int count = 0;
        while (0 != aMap.size()) {
            ++count;
            System.gc();
        }
        System.out.println("Took " + count
                + " calls to System.gc() to result in weakHashMap size of : "
                + aMap.size());
    }
}
```

Output: Took `20 calls to System.gc()` to result in `aMap size` of : 0.

`WeakHashMap` has only weak references to the keys, not strong references like other `Map` classes. There are situations which you have to take care when the value or key is strongly referenced though you have used `WeakHashMap`. This can avoided by wrapping the object in a **WeakReference**.

```
import java.lang.ref.WeakReference;
import java.util.HashMap;

public class Test {

    public static void main(String args[]) {
        HashMap<Employee, EmployeeVal> map =
                      new HashMap<Employee, EmployeeVal>();
        WeakReference<HashMap<Employee, EmployeeVal>> aMap =
                       new WeakReference<HashMap<Employee, EmployeeVal>>(
                map);

        map = null;

        while (null != aMap.get()) {
            aMap.get().put(new Employee("Vinoth"),
                    new EmployeeVal("Programmer"));
            System.out.println("Size of aMap " + aMap.get().size());
            System.gc();
        }
        System.out.println("Its garbage collected");
    }
}
```

Soft References.

`Soft Reference` is slightly stronger that weak reference. Soft reference allows for garbage collection, but begs the garbage collector to clear it only if there is no other option.

The garbage collector does not aggressively collect softly reachable objects the way it does with weakly reachable ones -- instead it only collects softly reachable objects if it really "needs" the memory. Soft references are a way of saying to the garbage collector, "As long as memory isn't too tight, I'd like to keep this object around. But if memory gets really tight, go ahead and collect it and I'll deal with that." The garbage collector is required to clear all soft references before it can throw `OutOfMemoryError`.

---

https://www.ibm.com/developerworks/java/library/j-jtp11225/

```
import java.lang.ref.ReferenceQueue;
import java.lang.ref.WeakReference;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.Random;
```

```java
import java.util.WeakHashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class A {
	public ExecutorService exec = Executors.newFixedThreadPool(5);
	public Map<Task, TaskStatus> taskStatus = 
Collections.synchronizedMap(new WeakHashMap<Task, TaskStatus>());
	private Random random = new Random();

	private enum TaskStatus {
		NOT_STARTED, STARTED, FINISHED
	};

	private class Task implements Runnable {
		private double[] numbers = new double[1000000];

		public void run() {
			int[] temp = new int[random.nextInt(10000)];

			doSomeWork();
			taskStatus.put(this, TaskStatus.FINISHED);
		}

		private void doSomeWork() {
			for (int i = 0; i < 10; i++) {
				Math.sin(30.0 / i * Math.PI);
			}
			try {
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
		}
	}

	public Task newTask() {
		Task t = new Task();
		taskStatus.put(t, TaskStatus.NOT_STARTED);
		exec.execute(t);
		return t;
	}

	class B {
		private double[] numbers = new double[10000];
	}

	public static void main(String[] args) throws InterruptedException {
		Thread.sleep(4000);

		A a = new A();
		for (int i = 0; i < 1000; i++) {
			a.newTask();
			System.out.println("status" + a.taskStatus.size());
		//	System.gc();
			System.out.println(i);
		}
		//a.exec.shutdown();
		// ReferenceQueue<B> rq = new ReferenceQueue<B>();
		//
```

```
            // Thread t = new Thread(()-> {System.gc();});
            // t.start();
            // for (int i = 0; i < 1000; i++) {
            // WeakReference<B> b = new WeakReference<B>(new B(), rq);
            // }
      }
}
```