GC Algorithm and Story of WeakReference

<http://www.codeproject.com/Articles/123448/GC-Algorithm-and-Story-of-WeakReference>

By **[Abhishek Sur](http://www.codeproject.com/script/Membership/View.aspx?mid=4293807)**, 31 Oct 2010

Top of Form

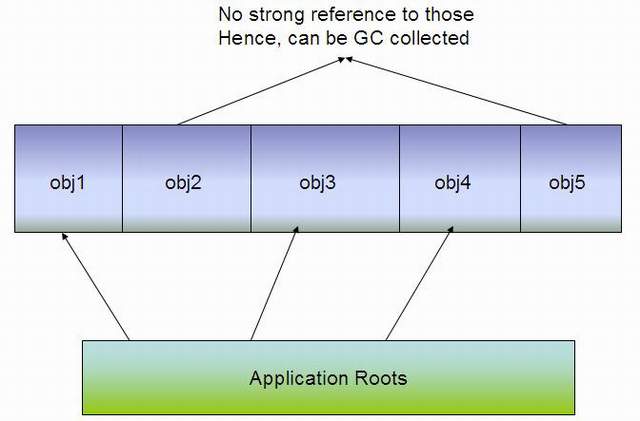
* [**Download test sample - 17.18 KB**](http://www.codeproject.com/KB/dotnet/WeakReference/WeakReferenceTestSample.zip)

Introduction

We all know that .NET objects deallocate memory using Garbage Collection. Garbage collection is a special process that hooks in to the object hierarchy randomly and collects all the objects that are not reachable to the application running. Let us make Garbage collection a bit clear before moving to the alternatives.

GC Algorithm

In .NET, every object is allocated using **Managed Heap**. We call it managed as every object that is allocated within the .NET environment is in explicit observation of GC. When we start an application, it creates its own address space where the memory used by the application would be stored. The runtime maintains a pointer which points to the base object of the heap. Now as the objects are created, the runtime first checks whether the object can be created within the reserved space, if it can it creates the object and returns the pointer to the location, so that the application can maintain a **Strong** **Reference** to the object. I have specifically used the term **Strong** **Reference**for the object which is reachable from the application. Eventually the pointer shifts to the next base address space.



When GC strikes with the assumption that all objects are garbage, it first finds all the **Strong** **Reference**s that are global to the application, known as Application Roots and goes on object by object. As it moves from object to object, it creates a Graph of all the objects that it finds from the application Roots, such that every object in the Graph is unique. When this process is finished, the Graph will contain all the objects that are somehow reachable to the application. Now as the GC already identified the objects that are not garbage to the application, it goes on Compaction. It linearly traverses to all the objects and shifts the objects that are reachable to non reachable space which we call as **Heap Compaction**. As the pointers are moved during the Heap compaction, all the pointers are reevaluated again so that the application roots are pointing to the same **reference** again.

Is **WeakReference** an Exception?

On each GC cycle, a large number of objects are collected to release the memory pressure of the application. As I have already stated, it finds all the objects that are somehow reachable to the Application Roots. The **reference**s that are not collected during the Garbage Collection are called **Strong** **Reference**, as by the definition of **StrongReference**, the objects that are reachable to the GC are called **Strong** **Reference** objects.

This creates a problem. GC is indeterminate. It randomly starts deallocating memory. So say if one has to work with thousand bytes of data at a time, and after it removes the **reference**s of the object it had to rely on the time when GC strikes again and removes the **reference**. You can use GC.Collect to request the GC to start collecting, but this is also a request.

Now say you have to use the large object once again, and you removed all the **reference**s to the object and need to create the object again. Here comes huge memory pressure. So in such a situation, you have:

1. Already removed all **reference**s of the object
2. Garbage collection didn't strike and removed the address allocated
3. You need the object again

In such a case, even though the object is still in the application memory area, you still need to create another object. Here comes the use of **WeakReference**.

Types of **WeakReference**

**WeakReference** can be of two types:

* **Short**: Short **WeakReference** loses the **reference** when the GC is collected. In our case, we have used short **WeakReference**.
* **Long**: Long **WeakReference** is retained even when the objects Finalize method is called. In this case, the object state cannot be determined. We pass trackResurrection to true in the constructor of**WeakReference** which defaults false, to track the object even though the Finalize method is called.

A **WeakReference** object takes the **Strong** **Reference** of an object as argument which you can retrieve usingTarget property. Let us look into the example:

To demonstrate the feature, let's create a class which uses a lot of memory.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/123448/GC-Algorithm-and-Story-of-WeakReference)

public class SomeBigClass : List<string>

{

public SomeBigClass()

{

this.LoadBigObject();

}

private void LoadBigObject()

{

for (int i = 0; i < 100000; i++)

this.Add(string.Format("String No. {0}", i));

}

}

Clearly the SomeBigClass is a list of 100000 strings. The code looks very straight forward, as I have just created an alternative to define List<string>. Now let's create another class to show the actual implementation of the **WeakReference** class.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/123448/GC-Algorithm-and-Story-of-WeakReference)

public class **WeakReference**Usage

{

**WeakReference** **weak**ref = null;

private SomeBigClass \_somebigobject = null;

public SomeBigClass SomeBigObject

{

get

{

SomeBigClass sbo = null;

if (**weak**ref == null)

*//When it is first time or object* ***weak****ref is collected.*

{

sbo = new SomeBigClass();

this.**weak**ref = new **WeakReference**(sbo);

this.OnCallBack("Object created for first time");

}

else if (**weak**ref.Target == null) *// when target object is collected*

{

sbo = new SomeBigClass();

**weak**ref.Target = sbo;

this.OnCallBack("Object is collected by GC,

so new object is created");

}

else *// when target object is not collected.*

{

sbo = **weak**ref.Target as SomeBigClass;

this.OnCallBack("Object is not yet collected,

so reusing the old object");

}

this.\_somebigobject = sbo; *//gets you a* ***strong******reference***

return this.\_somebigobject;

}

set

{

this.\_somebigobject = null;

}

}

# region GarbageEvent

public event Action<string> CallBack;

public void OnCallBack(string info)

{

if (this.CallBack != null)

this.CallBack(info);

}

# endregion

}

In the above class, we define a **reference** of **WeakReference** as **weak**ref, which holds the object ofSomeBigClass. Now the property SomeBigClass has little logic defined within it. It uses the existing**WeakReference**.Target to fetch the existing object. If the Target is null, the object will again be recreated and stored within the **WeakReference** Target again.

**WeakReference** serves as an exception to the existing GC algorithm. Even though the object is reachable from the application, it is still left for GC collection. So if GC strikes, it will collect the object of SomeBigClass and the**WeakReference**.Target will lose the **reference**.

Demonstration of the Code

To demonstrate the class, let's create a Console application and create the object of **WeakReference**Usage. Let the example class look like:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/123448/GC-Algorithm-and-Story-of-WeakReference)

static void Main(string[] args)

{

**WeakReference**Usage wru = new **WeakReference**Usage();

wru.CallBack += new Action<string>(wru\_CallBack);

while (true)

{

*//Access somebigclass*

foreach (string fetchstring in wru.SomeBigObject)

Console.WriteLine(fetchstring);

*//fetch complete.*

wru.SomeBigObject = null;

GC.Collect(); *// request to collect garbage.*

ConsoleKeyInfo info = Console.ReadKey();

if (info.Key == ConsoleKey.Escape)

break;

}

}

static void wru\_CallBack(string obj)

{

Console.WriteLine(obj);

Console.Read();

}

Here in the Main method, I have created an object of **WeakReference**Usage, and registered the callback so that whenever we try to retrieve the object, the message will be displayed in the console.

By setting:

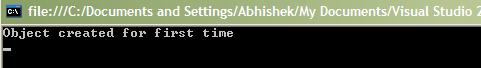
http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/123448/GC-Algorithm-and-Story-of-WeakReference)

wru.SomeBigObject = null;

GC.Collect(); *// request to collect garbage.*

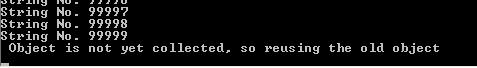
It will destroy the **strong** application **reference** and hence the object will be exposed for Garbage Collection. The call GC.Collect will request the garbage collection to collect.

Thus on first run, you will see:



The object is created for the first time.

weakrefdemo1.JPG



After you fetch all the data, you might either receive the second message, saying that the object is not yet collected and the object is fetched from the existing **WeakReference**, or if you wait for a long time, you might receive the 3rd message which says that the object is collected by GC and object is recreated.

Bottom of Form