# ANTS Memory Profiler 6.0

August 2010

Note: these pages apply to a version of
this product that is not the current released version.

For the latest support documentation, please see
http://documentation.red-gate.com

**redgate**®

**ingeniously simple tools**

# Contents

# Getting started

ANTS Memory Profiler enables you to profile memory usage of applications written in any of the languages available for the .NET Framework, including Visual Basic .NET, C#, and Managed C++. This is useful, for example, to improve memory usage by identifying the objects and classes that use most memory, and objects that remain live the longest.

You can use ANTS Memory Profiler to profile .NET desktop applications, ASP.NET web applications hosted in Internet Information Services (IIS) or the ASP.NET Development Server, .NET Windows services, COM+ server applications, Silverlight 4 or later applications, and XBAPs. In addition, you can profile applications that host the .NET Runtime, for example Visual Studio .NET plug-ins.

You can use ANTS Memory Profiler with the following versions of the .NET Framework:

- 1.1 (32-bit applications only)
- 2.0 (32-bit or 64-bit applications)
- 3.0 (32-bit or 64-bit applications)
- 3.5 (32-bit or 64-bit applications)
- 4.0 (32-bit or 64-bit applications)

Some filters and functionality are not available when profiling .NET 1.1 applications, see the List of object filters for more details.

## ANTS Memory Profiler: step-by-step

The exact procedure you use to find your memory problem depends on the type of problem you think you have. In all cases, there are three general steps:

1. Set up (page 18) the program that you want to profile.

2. Take one or more memory snapshots.

3. Analyze the profiling results.

# Types of memory problem

It is possible to categorize most types of memory problem you encounter; this may help you start to locate and resolve your problem.

- **Managed memory leaks**
  The typical symptom of a memory leak is that the performance degrades while the program runs but this recovers on restart and then degrades again. The amount of bytes in the .NET heap increases. Read more...

- **Unmanaged object leaks**
  The performance degrades while the program runs but this recovers on restart and then degrades again. The number of private bytes (the amount of real and paged memory requested by the program) in use often increases at a greater rate than the number of bytes in the .NET heap. Read more...

- **Fragmentation of the large object heap**
  The main symptom of fragmentation is that *OutOfMemoryException*s are thrown even though there is sufficient space on the .NET heap. Read more...

- **Large RAM footprint**
  Some programs need to use a lot of memory. There is little specific advice that can be given about this, since the best approach to reducing the amount of memory used will depend on your program's requirements. Using a profiler may give you suggestions for where to optimize code by showing which parts of your program are using the most memory.

## Further information

- Identify and prevent memory leaks in managed code (http://msdn.microsoft.com/en-gb/magazine/cc163491.aspx)

- Video tutorial (http://www.red-gate.com/products/ants_memory_profiler/DOTNET_Memory_Management/index.html)

- Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework (http://msdn.microsoft.com/en-us/magazine/bb985010.aspx)

- Memory Management in .NET (http://www.c-sharpcorner.com/UploadFile/tkagarwal/MemoryManagementInNet11232005064832AM/MemoryManagementInNet.aspx)

If you leave objects referenced when they are no longer needed, the amount of memory used will increase.  The .NET Garbage Collector (GC) can only free memory if an object (or an object's ancestor) has no reference pointing to it.  Therefore, if an object which is not needed is still referenced, it will not be freed.  References may be made from one object to another (through a member variable), from global or static variables, or from local variables in currently running methods.

A common example of memory leaks caused by this effect might be when a dialog box is displayed and a reference is made to it from another part of the user interface.  If this reference is not destroyed, the dialog box and its contents will remain in memory even after it has been closed.

The typical symptom of a memory leak is that the performance degrades while the program runs but the amount of memory used recovers on restart and then degrades again.

Managed .NET code may interoperate with unmanaged code when some COM objects or C-style DLLs are invoked.

Leaks caused by unmanaged objects are more complex to identify, because ANTS Memory Profiler cannot provide detailed information about unmanaged usage. You can compare the different performance counters provided by ANTS Memory Profiler, however, to give some ideas about where to look for these leaks.

The main indication of an unmanaged memory leak is when the number of private bytes (the amount of real and paged memory requested by the program) increases while the number of bytes in the .NET heap does not grow as quickly. You may be able to find the source of the problem by looking for .NET objects whose instance count is increasing, or by looking for objects on the finalizer queue that have not been disposed.

**Further information**

- Eric Smith has written a blog post describing his approach to finding an unmanaged leak
  http://esmithy.net/2007/06/25/finding-unmanaged-memory-leaks/
  (http://esmithy.net/2007/06/25/finding-unmanaged-memory-leaks/)

Although it is not technically a memory leak, fragmentation of the Large Object Heap (LOH) can create similar effects.  The LOH is used to store objects exceeding 85kB (most commonly arrays of objects or long strings).  The .NET runtime will only run a Garbage Collection (GC) on the LOH during a full Garbage Collection (that is, when a Generation 2 collection takes place on the Small Object Heap).  After a collection, the LOH is never compacted because this would seriously reduce the performance of your program.

This means that if an object on the LOH is removed during a GC, free space will exist on the LOH which will be the same size as the deleted object. The problem is that if you try to allocate a new object which is larger than the deleted object (even by one byte), the CLR will add it to the end of the LOH by making the LOH larger if necessary.  Eventually, the LOH will become too large and an OutOfMemoryException is thrown, even though there may be plenty of free space within the heap.

**Further Information**

- The Dangers of the Large Object Heap
  http://www.simple-talk.com/dotnet/.net-framework/the-dangers-of-the-large-object-heap/ (http://www.simple-talk.com/dotnet/.net-framework/the-dangers-of-the-large-object-heap/)

- The MSDN article 'Large Object Heap Uncovered'
  http://msdn.microsoft.com/en-us/magazine/cc534993.aspx (http://msdn.microsoft.com/en-us/magazine/cc534993.aspx)

- Understanding the Garbage Collector
  http://www.simple-talk.com/dotnet/.net-framework/understanding-garbage-collection-in-.net/ (http://www.simple-talk.com/dotnet/.net-framework/understanding-garbage-collection-in-.net/)

# Strategies for memory profiling

This topic contains guidelines and recommendations about common memory profiling scenarios, including some heuristics about memory usage patterns that might indicate a memory problem. Of course, every application is different, so these guidelines only outline strategies for memory profiling; to apply these strategies you will need a good understanding of your application.

## Checking for memory problems

The following steps outline how to carry out a systematic check of memory usage, to determine whether there are any memory problems.

1. Before you start, work out a plan for what functionality or states you want to check. You will need to be methodical about taking snapshots at the appropriate times to check these states, and you will need to have a good understanding of the expected memory usage - so you can identify unexpected memory usage.

2. Start ANTS Memory Profiler and start profiling your application. Take a snapshot of the application in each of the states you want to compare. Depending on what you are trying to discover, it may be sufficient to take a snapshot before and after performing actions, or you may need to take additional snapshots during use, so that you can analyze memory usage throughout.

3. It may help with your analysis if you change the name of the snapshots so that they are easier to recognize.

4. For each state that you want to check on, select a baseline and current snapshot in the **Snapshots** bar. Alternatively, you can check on memory usage in a single state in isolation: from the **Current** list, select the snapshot you are interested in; from the **Baseline** list, select *No baseline*.

   ♦ Look at each snapshot to understand where memory is being used in each state.

      Find out more about how to identify what is using most memory

   ♦ Compare pairs of snapshots to check for memory leaks.

      Read suggested strategies for finding memory leaks

   ♦ Use the object filters to check for common indicators of memory leaks.

      Find out more about using filters

   ♦ Look for fragmentation problems on the large object heap

      Read tips on identifying fragmentation problems on the large object heap

The following steps outline how to find out what classes are using most memory in your application.

1.  Start ANTS Memory Profiler and start profiling your application. Get the application into the state in which you are interested in its memory, and then take a memory snapshot.

2.  On the class list, look for the largest classes, or the classes with the highest number of instances. For this analysis, you are going to explore memory usage by following references to a class, so it is not important if you do not recognize these classes.

3.  Select a class and display the class reference explorer. Use the explorer to look at what is keeping your selected class in memory.

    Start by looking at the class with the highest number of direct references to objects in your selected class (this is the class at the top of the graph, to the left of your selected class). Click on the class to show classes that refer to it, and then continue to follow the chain of references to understand what is keeping the instances of your selected class in memory.

    For example, *System.String* is often the largest class; using the class reference explorer you can find out what is keeping strings in memory.

# Finding and fixing a memory leak

The following examples outline some approaches to finding a memory leak for some common scenarios.

We recommend that you make a note of the steps you take when you are looking for a memory leak, so that you can perform the same actions later to check that you have fixed the leak.

The snapshots and analysis you perform depend on the functionality of your application.

## Example A: opening and closing a dialog box

This scenario is the most straightforward, so it is recommended as the preferred way of finding a memory leak.

1. Start ANTS Memory Profiler and start profiling your application. Get the application into the state in which you are interested in its memory.

2. Take two snapshots, so you can compare memory usage before and after the action that you believe leaks memory:

   a. Take the first snapshot.

   b. Perform an action that you believe causes a leak; then perform the actions that should clean up any objects created by the first action.

      For example, open a dialog box, change some settings, and then close the dialog box.

   c. Take a second snapshot.

   Objects created by the action should be cleaned up before the second snapshot, so any new objects created in the second snapshot are likely to indicate a memory leak.

3. Apply the **Only new objects** filter to show only the classes with new instances in the second snapshot.

4. On the class list, look for classes with a high positive value in the **Instance Diff** or **Size Diff** column. These values indicate the classes responsible for increased memory usage in your second snapshot, so they are good indicators of the likely cause of a memory leak. At this stage, we recommend that you look at *all* classes (not just the classes you recognize): although your own classes may be responsible for the memory leak, the symptoms of the leak may be increased usage in other classes, such as *System.String*.

5. Look at instances of classes with unexpectedly high growth in size or number of instances:

   ♦ If the class that looks interesting is one you recognize, look at instances on the instance list.

   ♦ If the class that looks interesting is not one you recognize, use the class reference explorer to navigate along the chain of references to objects in this class, until you reach a class you recognize. Next, look at instances of that class on the instance list.

6. On the instance list, look for objects with a high value in the **Distance from GC Root** column.

   Often, leaked objects are found at a greater distance from their nearest GC root because all the obvious, shorter chains of reference from a GC root to an object have been broken already.

7. Show the object retention graph for the object that looks interesting. Follow chains of references up the graph to identify objects keeping your object in memory unexpectedly.

8. When you find an unexpected reference, modify your code to break the reference, and then profile the application again to check that the problem is fixed.

**Example B: populating and clearing a list**

1. Start ANTS Memory Profiler and start profiling your application. Get the application into the state in which you are interested in its memory.

2. Take two snapshots, so you can compare memory usage before and after the action that you believe leaks memory:

   a. Perform the action that you believe causes a leak. For example, populate a list with data.

   b. Take the first snapshot during or immediately after this action - that is, before any clean-up happens.

   c. Perform the action that should clean up the objects created by the first action. For example, clear the data from your list.

   d. Take a second snapshot.

   Objects should be cleaned up between snapshots, so any remaining objects were probably created by the first action.

3. Apply the **Only surviving objects** filter to show only classes that exist in both snapshots.

4. On the class list, look for classes with a high value in the **Live Instances** or **Live Size** column. These values indicate the classes responsible for memory usage in your snapshots. At this stage, we recommend that you look at *all* classes (not just the classes you recognize): although your own classes may be responsible for the memory leak, the symptoms of the leak may be increased usage in other classes, such as *System.String*.

5. Look at instances of the class with unexpectedly high size or number of instances:

   ♦ If the class that looks interesting is one you recognize, look at instances on the instance list.

   ♦ If the class that looks interesting is not one you recognize, use the class reference explorer to navigate along the chain of references to objects in this class, until you reach a class you recognize. Next, look at instances of that class on the instance list.

6. On the instance list, look for objects with a high value in the **Distance from GC Root** column. Often, leaked objects are found at a greater distance from their nearest GC

root because all the obvious, shorter chains of reference from a GC root to an object have been broken already.

7. Show the object retention graph for the object that looks interesting. Follow chains of references up the graph to identify objects keeping your object in memory unexpectedly.

8. When you find an unexpected reference, modify your code to break the reference, and then profile the application again to check the problem is fixed.

**Example C: general strategy recommendation**

This way of finding a memory leak is recommended:

- if you are not sure what functionality or actions are causing the memory leak

- for applications where you expect memory to be constant, but instead memory usage increases slowly

- for applications that do not have functionality that you can manually execute and then clean up

1. Start ANTS Memory Profiler and start profiling your application.

2. Run your application and monitor memory usage on the timeline. When memory usage starts to increase, take several snapshots (the frequency and number of snapshots you need to take depends on your application and how rapidly memory increases).

3. Select two snapshots to compare.

4. If there are classes that you expect to be large or increasing in size, apply the **Never referenced by an instance of *class/interface*** filter to remove these classes from the results.

5. On the class list, look for classes with a high value in the **Size Diff** column. This value indicates the growing classes, so it is a good indicator of a memory leak. At this stage, we recommend that you look at *all* classes (not just the classes you recognize): although your own classes may be responsible for the memory leak, the symptoms of the leak may be increased usage in other classes, such as *System.String*.

6. Look at instances of the class with the largest difference in size between snapshots:

   ♦ If the class that looks interesting is one you recognize, look at instances on the instance list.

   ♦ If the class that looks interesting is not one you recognize, use the class reference explorer to navigate along the chain of references to objects in this class, until you reach a class you recognize. Next, look at instances of that class on the instance list.

7. On the instance list, apply the **Only surviving objects** filter to show only the classes that exist in both snapshots, and then look for high values in the **Live Size** column. This identifies the largest objects, which have stayed in memory for longest, so may indicate the cause of a leak.

8. Show the object retention graph for the largest object that should not be in memory. Follow chains of references up the graph to identify objects keeping your object in memory unexpectedly.

9. When you find an unexpected reference, modify your code to break the reference, and then profile the application again to check the problem is fixed.

# Checking that a memory leak is fixed

The following steps outline how to check that a memory leak you previously identified is now fixed:

1. Repeat the steps you used to find the memory leak. If you are looking for a particular class or object, it may be useful to use the  ⌕ find box to locate the class or object you are interested in.

2. If you have fixed the leak, the objects should not be in memory.

   If an unexpected instance *does* seem to still be in memory, display the object on the object retention graph, and check whether it is on the finalizer queue (clear the **Hide finalizer queue GC roots** option on the bar above the graph). If your object is on the finalizer queue, take another snapshot and check again: the object may be removed when the garbage collector runs.

The large object heap charts show an overview of memory availability for the large object heap.

- **Free space on all .NET heaps** shows the total amount of memory reserved for .NET but not currently used.

- **Largest free block** shows the size of the largest block of unused memory that is currently reserved for .NET.

- **Max. size of new object** shows an estimate of the size of the largest object that can be allocated. This value is always at least the size of the largest free block, and may be larger where it is possible to extend the large object heap by reserving more memory.

Note: This information is not available when you profile a .NET 1.1 application.

**More information about the large object heap**

CLR Inside Out: Large object heap uncovered
http://msdn.microsoft.com/en-us/magazine/cc534993.aspx
(http://msdn.microsoft.com/en-us/magazine/cc534993.aspx)

Dangers of the large object heap
http://www.simple-talk.com/dotnet/.net-framework/the-dangers-of-the-large-object-heap/ (http://www.simple-talk.com/dotnet/.net-framework/the-dangers-of-the-large-object-heap/)

**Identifying fragmentation problems**

The following states can indicate fragmentation problems:

- The **Largest free block** size is small, but the **Free space on all .NET heaps** size is large. Although there is plenty of unused memory, the blocks of contiguous unused memory are small, so there may be a danger of running out of memory because the memory is not used efficiently.

- The **Max. size of new object** size is small, but the **Free space on all .NET heaps** size is large. Although there is plenty of unused memory, it may not be possible to allocate memory for new large objects, because available contiguous memory slots are small and it is not possible to allocate more memory to increase the size of the large object heap.

- When **Max. size of new object** is smaller than the amount of memory that your application tries to allocate for a new object, the application will fail with an *OutOfMemoryException*.

To investigate objects on the large object heap further, apply the **Objects on the large object heap** filter. This can be useful, for example, to find out which objects are currently in memory, and why they are in memory.

## Solving large object heap fragmentation

The best approach to use to solve fragmentation problems will depend on the exact nature of your program:

- Split arrays into smaller units so that they remain below 85kB (and so are never allocated on the LOH).

- Alternatively, you can allocate the largest and longest-living objects first (if your objects are files which are queued for processing, for example).

- In some cases, it may be that periodically stopping and restarting the program is the only option.

# Setting up the profiler

To profile an application, you must first set up a profiling session. A session specifies:

- The application type, location, and options for the application you want to profile.
- The profiling mode, which determines the level of detail gathered by the profiler while your application is running.
- The method used to calculate timing values (CPU time or wall-clock time).
- The performance counters to display on the timeline.

When you start ANTS Memory Profiler, the **ANTS Memory Profiler Settings** dialog box is automatically displayed; if ANTS Memory Profiler is already running, click **New Profiling Session** on the **File** menu.

The **Application Settings** tab displays the settings for the last profiling session you ran. The settings available depend on the selected application type, and may differ from those illustrated above.

The **Charting Options** tab enables you to choose which performance counter values to display on the timeline for the new profiling session.

**To set up and run a profiling session:**

1. On the **ANTS Memory Profiler Settings** dialog box, complete the details on the Application Settings tab.
2. Choose which performance counters to monitor during profiling, using the Charting Options tab.
3. Click **Start Profiling**

   The timeline is displayed at the top of the main ANTS Memory Profiler window, and the application you want to profile is automatically started. Status text at the bottom-left of the main window indicates what ANTS Memory Profiler is doing during the profiling session.

   The timeline starts displaying performance-counter data and events in near-real time. There may be a slight delay between starting a profiling session and seeing the first performance-counter data appear on the timeline.

4. To display profiling results, do one of the following:

   - Drag a region on the timeline.

     Profiling data is summarized and displayed for the selected time period only. Your application will continue running and profiling will continue.

   - Click **Stop Profiling**.

     Your application will be closed. Profiling data is summarized and displayed for the entire profiling period.

   - Close your application.

     Profiling data is summarized and displayed for the entire profiling period.

You can continue working with the timeline to locate periods of interest during the execution of your application, and to display the associated profiling results.

Once you have displayed some profiling data, you can view and analyze it. For more information about the different ways you can do this, see Understanding the profiling results.

# Working with application settings

When you start ANTS Memory Profiler, or when you start a new profiling session, the ANTS Memory Profiler Settings dialog box is displayed.

## Application settings

On the **Application settings** tab, select the type of application you want to profile from the **Choose application type to profile** list. The settings available change depending on your selection:

- .NET executable
- ASP.NET web application (hosted in IIS or web development server)

  Note that special requirements apply to SharePoint.
- Silverlight 4 browser application
- Windows service
- COM+ server
- XBAP (XAML Browser Application)
- Attach to a .NET 4 process

## Charting options

Use the **Charting options** tab to choose which performance counters you would like to record. See Setting up performance counters for more information.

# Profiling SharePoint

ANTS Memory Profiler can profile managed code that runs on a Microsoft SharePoint server because Microsoft SharePoint 2007 is implemented as an ASP .NET web application.

Most users find that profiling a SharePoint server 'just works'.  If you encounter difficulties, the most likely cause is that ANTS Memory Profiler cannot read from the directory which SharePoint is writing data to. To fix this:

1. Create a temporary directory

2. If you are not on a sensitive system, allow full read/write access to this temporary directory to all users.  If you are on a sensitive system, ensuring that the local system account has read/write access should suffice.

3. Use **Control Panel** to add a new environment variable.  The variable must be called *RGIISTEMP* and the value is the path to the temporary directory you just created.

For more information, see Chris Allen's blog post, 'Profiling SharePoint with ANTS Performance Profiler 5.2 (http://www.simple-talk.com/dotnet/.net-tools/profiling-sharepoint-with-ants-performance-profiler-5.2/)' (The information in this post is also valid for ANTS Memory Profiler)

## If you experience problems

The instructions above will work for the majority of people but the security features in ASP.NET may cause problems on some systems.  In that case, follow the instructions below:

### Information you will need

Before you start setting up a SharePoint 2007 site for profiling, you must know:

- the URL for the site collection
- the TCP port it runs on
- the name of the primary site collection administrator
- the primary site collection administrator's password

for the site collection which you are profiling.

To find the name of the primary site collection administrator:

1. Open the SharePoint Central Administration website using the Start menu item.
2. Click the **Application Management** tab.
3. Under **SharePoint Site Management**, click **Site Collection Administrators**.
4. Select the name of the site collection hosting your web part from the dropdown list.

5.  Make a note of the account set in the **Primary Site Collection Administrator** box.
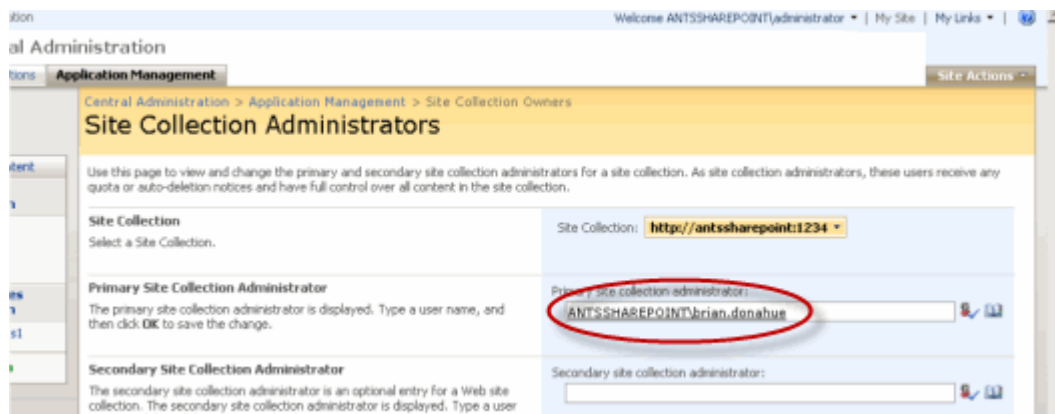


**Grant permissions**

The primary site collection administrator must have permission to launch an IIS 6 worker process. To grant this permission:

1.  Open **Administrative Tools** then open **Local Security Policy.**

2.  Under **Local Policies**, click **User Rights Assignment.**

3.  Double-click **Act as part of the operating system** and add the primary site collection administrator's account.
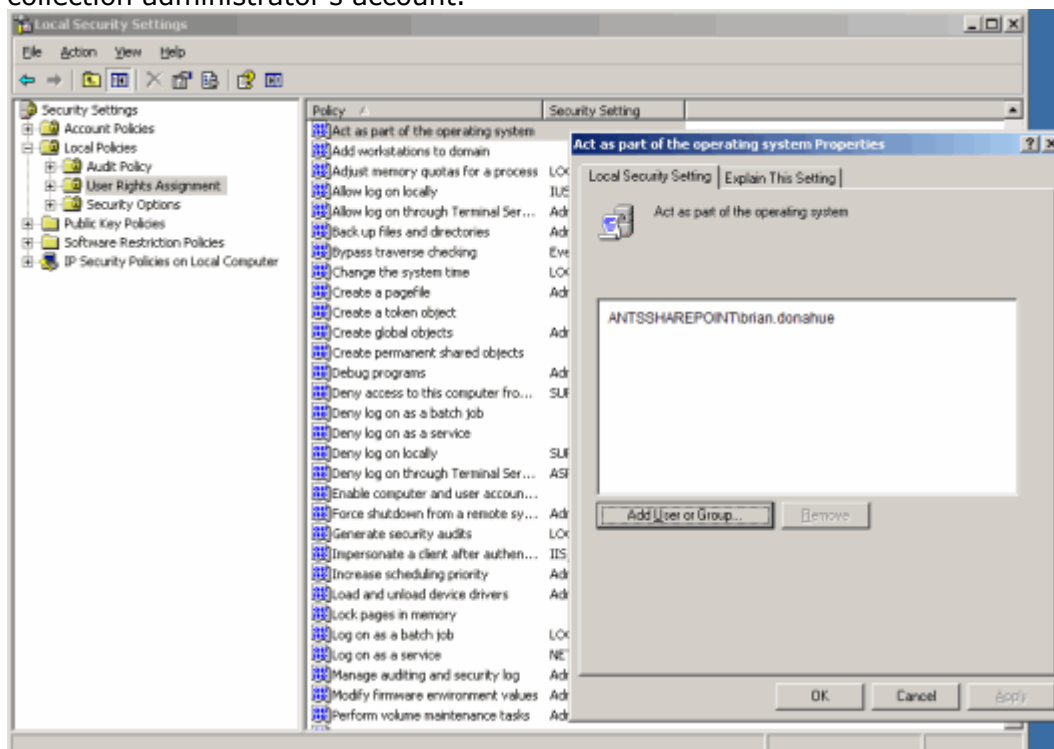


4.  Double-click **Impersonate a client after authentication** and add the primary site collection administrator's account.

5. Open a command prompt and run
   `gpupdate /force`
   to enforce the new settings.

6. Open **Administrative Tools** and go to **Computer Management**.

7. Under **Local Users and Groups**, open **Users**.

8. Double-click the primary site collection administrator's account and open the **Member Of** tab.

9. Add the *Administrators* group.
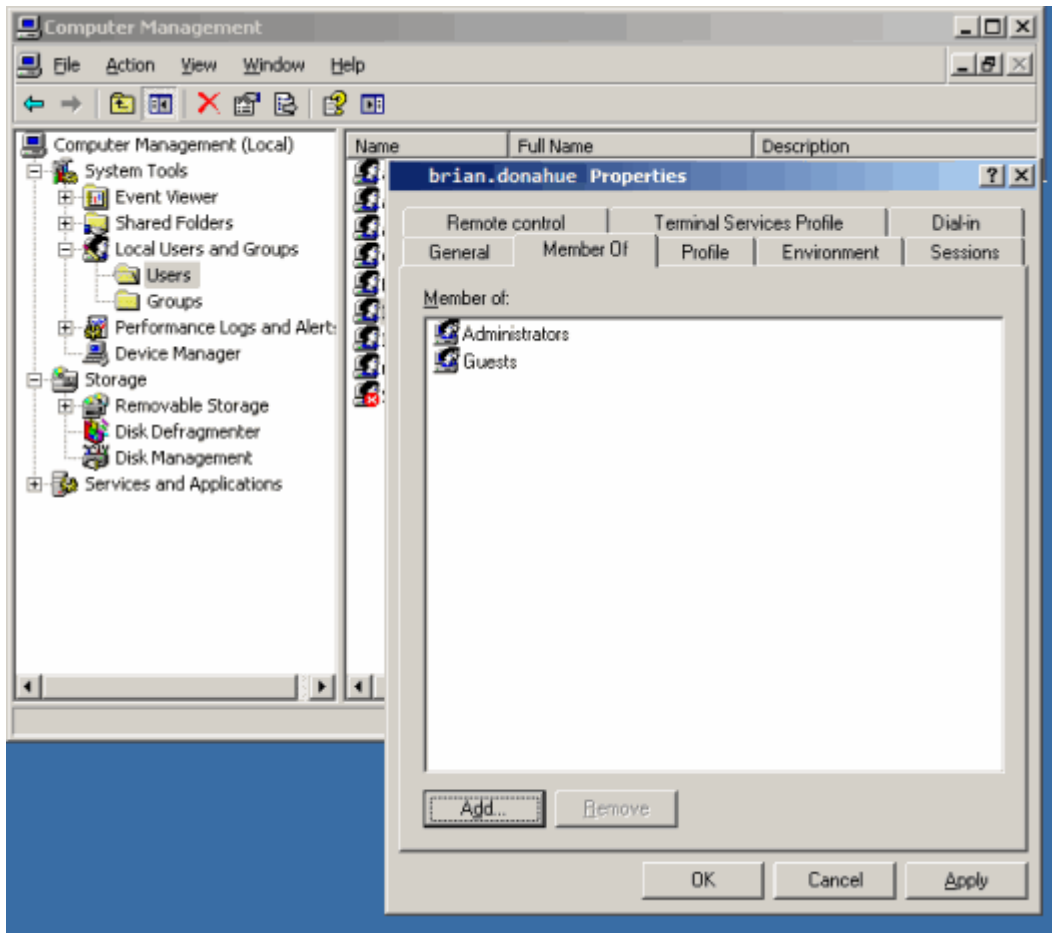


The *ANTS Memory Profiler 6.0 Service* must use the primary site collection administrator's account when it starts.  To configure this:

1. Open **Administrative Tools** then open **Services**.

2. Double-click the **ANTS Memory Profiler 6.0 Service**.

3. Click the **Log On** tab.

4. Select **This Account** and enter the primary site collection administrator's username and password.

5. Click **OK**.

6. If the status of **ANTS Memory Profiler 6.0 Service** is *Started*, right-click the service and click **Restart**.
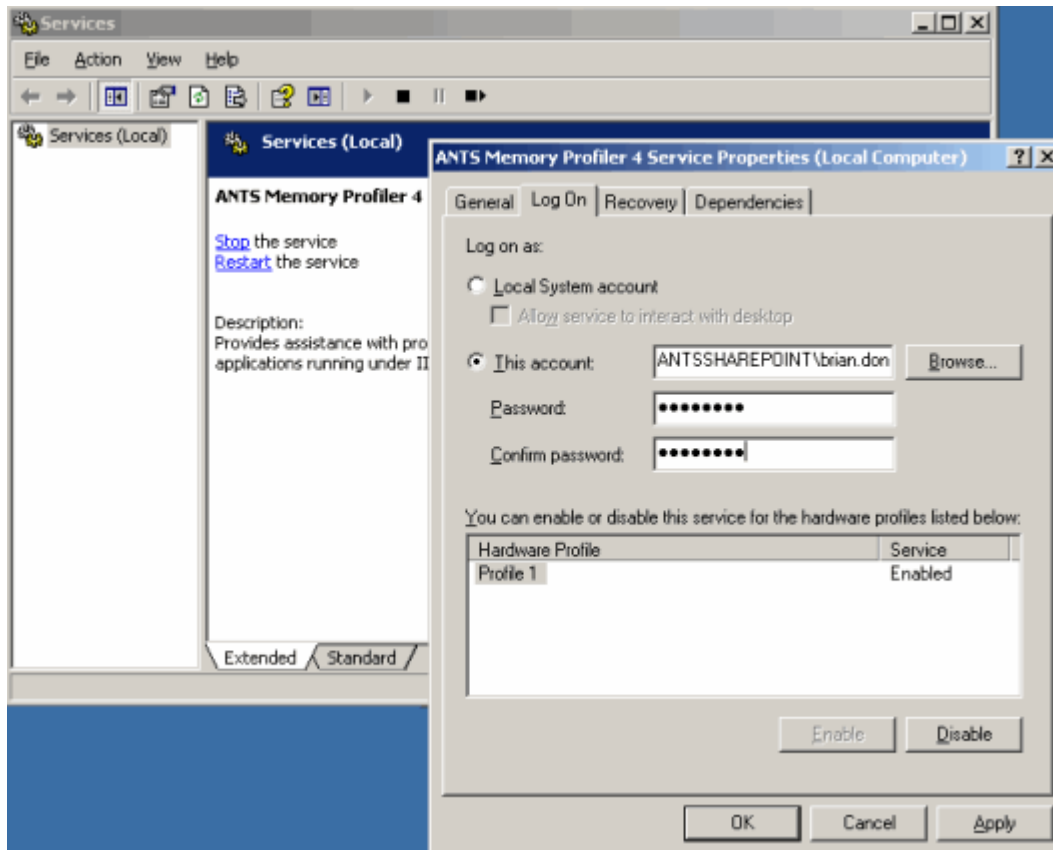


**Ensure that compilation will be in DEBUG configuration**

To profile a SharePoint collection, the ASP .NET compilation must be done in DEBUG configuration. This will allow ANTS Memory Profiler to locate the source code for any web parts or other extensions you have written for the site collection. DEBUG configuration will also relax some unmanaged code restrictions that prevent profiling and stop the site from timing out.

To set DEBUG configuration, you must know the physical path to the root of the site collection website.

To find this path:

1. Open **Administrative Tools**.
2. Open **Internet Information Server (IIS) Manager**.
3. Right-click the website containing the site collection then click **Properties**.

4. Open the **Home Directory** tab.
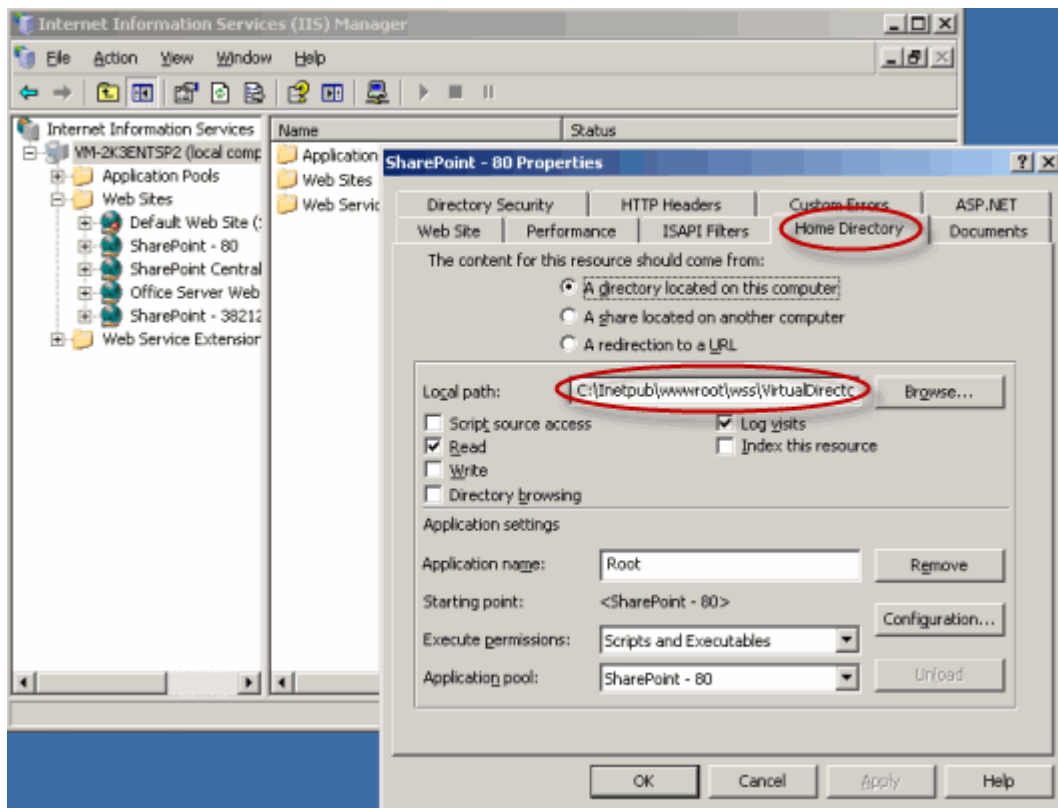


You must now locate and edit the web.config file for the site collection using notepad.exe (or other text-editor).

1. Use Windows Explorer to navigate to the site collection root's physical path.

2. Right-click the web.config file.

3. Open the web.config file using notepad.exe. Search for the text '***Debug***'.
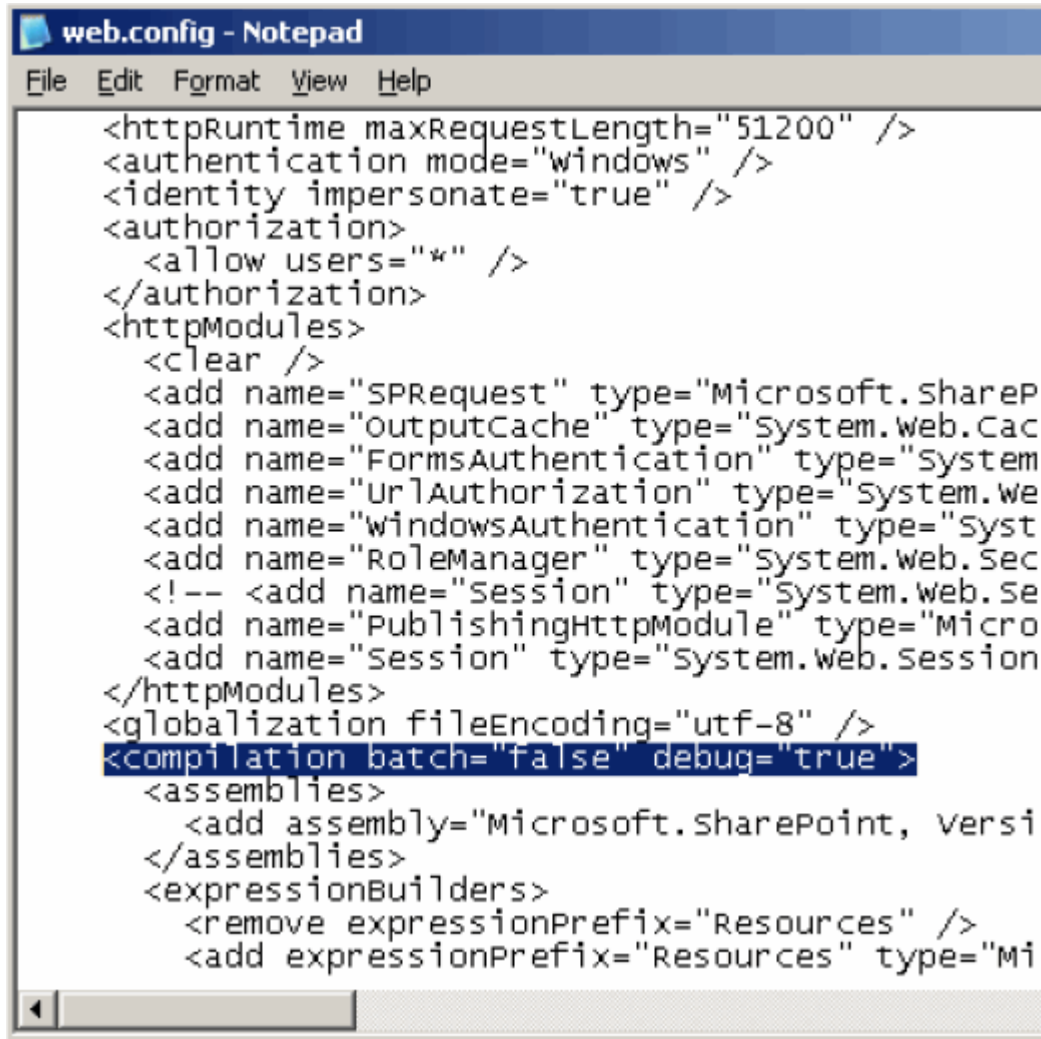
4. Change
   `Debug="False"`
   to
   `Debug="True"`
   and save the file.



**Copy PDBs and web part DLLs to the app_bin directory**

If you want to be able to filter out all methods except the ones run by your code when viewing the results, you must copy the relevant files into the site's app_bin directory. To do this:

1. Copy all PDB files and any web part DLLs used by your site to the Clipboard.

2. Use Windows Explorer to navigate to the site collection root's physical path.

3. Open the app_bin directory.

4. Paste all PDB files and any web part DLLs used by your site into this directory.

1. If the Internet Information Services Manager is not already displayed, open **Administrative Tools** and open **Internet Information Server (IIS) Manager**.

2. Stop the website.

3. Start ANTS Memory Profiler.

4. Under **Choose application type to profile**, click **ASP .NET Web Application (hosted in IIS)**.

5. Enter the path to the ASP .NET web application that hosts your site collection. The path should be in the following format:
   http://server:port/
   The server is the name of the local server and the port is the TCP port that the web application normally runs on. If the site collection is on the root virtual directory for the site, you must include the trailing slash is included.

6. To profile your application without restarting IIS select **Unused port** and enter a new port number.  If your application has port numbers hard-coded, this will not work and you must chose **Original port** instead.

7. Use your SharePoint 2007 Site Collection as normal. Any additions that you have coded, such as web parts and lists, will be reflected in the ANTS Memory Profiler results if these objects have been accessed.



---
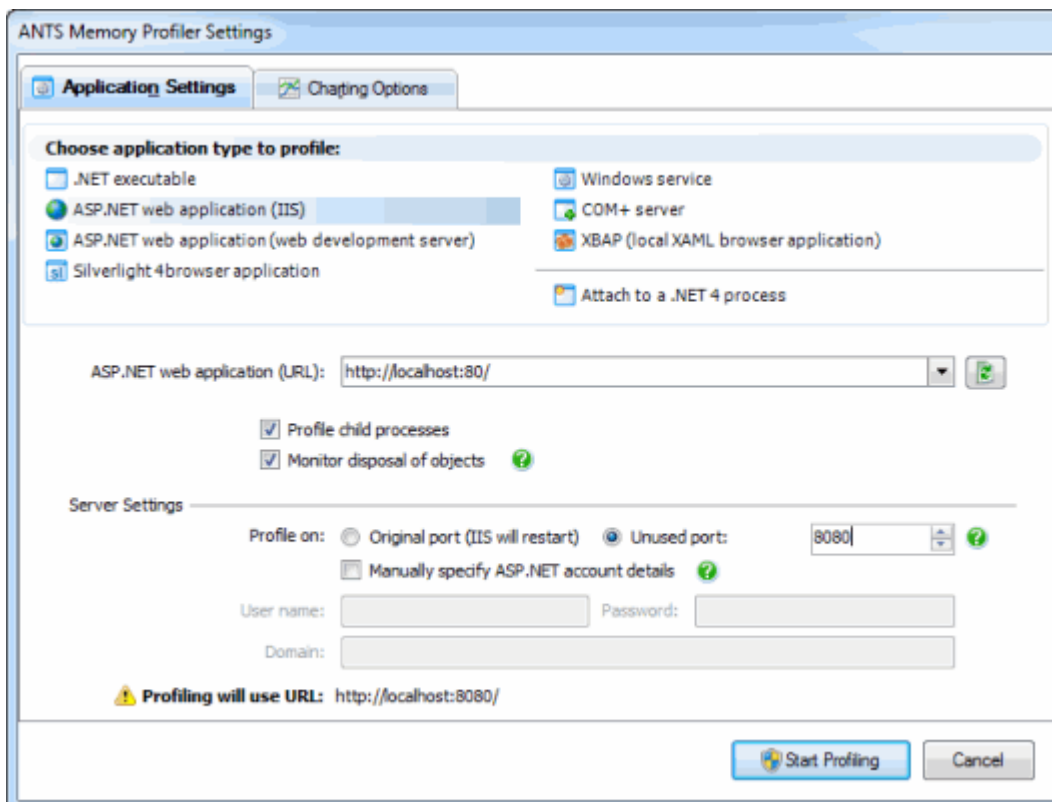
# Attaching to a running .NET 4 process

A list of all currently-running processes is shown. Select the .NET 4 process to profile. (Processes which are not compiled with the .NET 4 runtime are unavailable.) Note that you must disable concurrent garbage collection in the .NET 4 process.

Before you can attach ANTS Memory Profiler to a running .NET 4 process, you must first disable concurrent garbage collection in your application.  This allows ANTS Memory Profiler to monitor garbage collections, which it cannot do if garbage collections take place in a different thread.

Disabling concurrent garbage collection means that garbage collection will take place in the same thread as the application's thread. Note: This may affect the performance of your application; see the notes below. You can find a description of the concurrent garbage collector at http://msdn.microsoft.com/en-us/library/ee787088.aspx#concurrent_garbage_collection (http://msdn.microsoft.com/en-us/library/ee787088.aspx#concurrent_garbage_collection)

To disable concurrent garbage collection, follow these steps:

**If you are not attaching to a webservice running on the WebDev server:**

1. Create a new file, and copy the following XML into it:

```xml
<configuration>

    <runtime>

        <gcConcurrent enabled="false"/>

    </runtime>

</configuration>
```

2. Save this file in the same directory as the application's executable file.  The file must have the same name as the executable file, but a .config extension. (For example, if your application is *sample.exe*, the associated configuration file must be named *sample.exe.config*).

    It is not necessary to rebuild your executable. The .NET CLR will automatically load the configuration file at runtime.

3. Run your executable and then attach ANTS Memory Profiler to it (see Working with application settings).

4. When you have finished profiling, delete the configuration file.

**If you are attaching to a webservice running on the WebDev server:**

1. Open the *WebDev.WebServer.exe.config* file, located at *%ProgramFiles%/Common Files/microsoft shared/Dev Server/*, in a text editor or XML editor.

2. Under the `<runtime>` node, insert the following text:
   `<gcConcurrent enabled="false"/>`

3. Save the WebDev config file.

4. When you have finished profiling, delete the `<gcConcurrent>` node.

**Notes:**

- While concurrent garbage collection is disabled, your application will run faster but may pause occasionally to allow garbage collections to take place.

- While concurrent garbage collections are disabled, the total memory footprint of your application is fractionally increased. This is because the concurrent garbage collector would normally build a graph of unreachable objects on the other thread. This difference should not be significant enough to alter the results shown by ANTS Memory Profiler.

- You cannot attach to a .NET 4 process which is running in server garbage collection mode. If your configuration file contains `<gcServer enabled="true"/>`, delete this node.

ANTS Memory Profiler can monitor the values of a number of Windows performance counters while the application you are profiling is executing. The values of these counters are constantly updated on the timeline during profiling.

Choose the performance counters you want to monitor using the **Charting Options** tab on the **ANTS Memory Profiler Settings** dialog box.
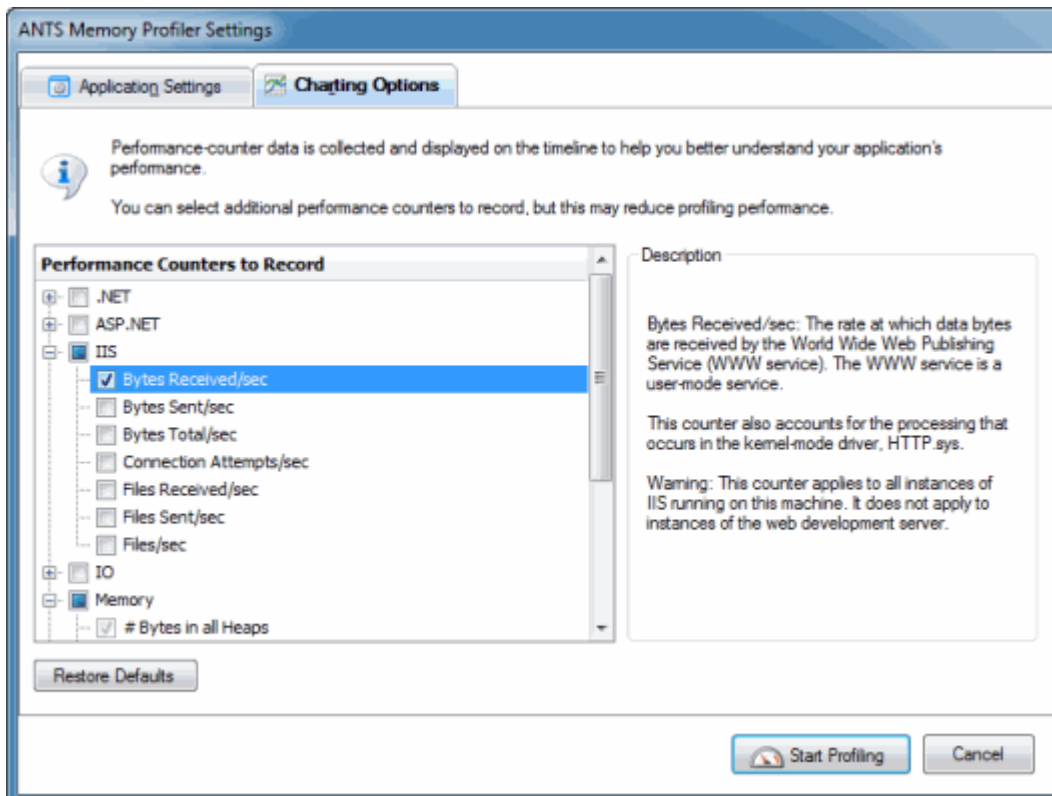


Not all performance counters may be appropriate for the application type you are profiling. You can find more information about individual performance counters in the **Description** box, including details about a counter's relevance to particular application types.

We recommend that you avoid adding more performance counters than you need, as each additional counter adds to the overhead introduced by the profiler. Adding too many counters may, therefore, cause your application to run substantially more slowly.

Private Bytes are shown by default in the timeline in ANTS Memory Profiler because it is considered the most useful measure for most kinds of memory problem. No single counter, however, reveals a complete picture of the memory requirements of your program.  The following table summarizes some of the main counters:

| Counter | Counts memory... | Including shared processes? (DLLs in memory, .NET runtime) | Including memory paged to disk? |
| --- | --- | --- | --- |
| **Private bytes** | ...asked for (even if not in use) | No | Yes |
| **Working set** | ...in use | Yes | No |
| **Virtual bytes** | ...in use | Yes | Yes |
| **Private working set*** | ...in use | No | No |

\* Windows Task Manager shows the private working set by default.  ANTS Memory Profiler does not show the private working set because it is only accurate immediately after a garbage collection.

**Adding custom performance counters**

You can add custom performance counters to the list of available counters in the **Charting Options** tab. To do this:

1. Close ANTS Memory Profiler.

2. Expose your performance counter to the Windows Performance Counter API using the *PerformanceCounter* and *PerformanceCounterCategory* classes of the *System.Diagnostics* namespace. (An example describing how to do this is given at http://msdn.microsoft.com/en-us/library/system.diagnostics.performancecounter.aspx (http://msdn.microsoft.com/en-us/library/system.diagnostics.performancecounter.aspx))

3. Create a new XML file as follows:
   ```
   <Counters>
     <Category Name="CategoryName">
       <Counter Category="CategoryName" Name="CounterName"
   Units="Measurement Units">
         <Instanced />
       </Counter>
     </Category>
   </Counters>
   ```
   Ensure that *CategoryName* and *CounterName* are the same as the names used for the *PerformanceCounterCategory* and *PerformanceCounter*. Remove the `<Instanced />` node if your counter collects data about the computer, not only the individual process. You can add multiple categories and counters in the same XML file.

4. Save the XML file as *UserCounters.xml* in *%LOCALAPPDATA%\Red Gate\ANTS Memory Profiler 6\*.

5. Restart ANTS Memory Profiler.

6. The counters that you defined are shown in the list on the **Charting Options** tab.

# Working with the timeline

The timeline is visible throughout a profiling session, and provides a frequently updated display of performance-counter values and instances of events related to the application you are profiling. You can use this overview of application activity to isolate memory-profiling results for specific time periods.



Timeline showing that memory is not released after a dialog is closed.

The timeline shows the values for a selection of Windows performance counters. You can choose which performance counters to display before you start profiling your application. See Setting up performance counters for more information.

Note: If you are profiling a Silverlight application, the heap size counters are not displayed in the timeline. This is due to a limitation in Silverlight. Click the **Session Overview** tab to show these values.

### Snapshots

While you are profiling your application, click the **Take Memory Snapshot** button in ANTS Memory Profiler to record memory usage at that time. You can name a snapshot to make it easier to identify. To name a snapshot, click 🖉 next to its name. The name you type will be shown on the snapshot's tooltip and in the *Baseline: / Current:* combo boxes beneath the timeline, if appropriate.

You can analyze memory usage by looking at a snapshot of memory at a specific point in time, or by comparing two snapshots. This topic outlines how to use the main ANTS Memory Profiler features to analyze memory usage.

Start by choosing an application to profile, and then take one or more snapshots, depending on the analysis you want to perform. ANTS Memory Profiler displays a summary of memory usage in your selected snapshots. Analyzing memory usage has the following main stages:

1. Snapshot analysis

2. Class analysis

3. Instance analysis

Before you begin, it may be useful to read background information about memory:

• Read suggested strategies for specific types of memory analysis

• Watch .NET memory management introduction video (http://www.red-gate.com/products/ants_memory_profiler/DOTNET_Memory_Management/index.html )

# 1. Snapshot analysis

Start by looking for classes that may indicate a memory leak, or classes with unexpectedly high memory usage. Use object filters to restrict your investigation to objects with specific characteristics or specific parts of the application.

## What to look for in the summary

The **summary** shows an overview of the main areas of memory usage; if you are comparing two snapshots, this includes an overview of the main differences in memory usage between the snapshots. In the summary or class list, look for classes with unexpectedly high memory usage, or a large increase in size between snapshots.

The large object heap data can be useful for identifying fragmentation problems. (For more information on this, see Finding LOH fragmentation)
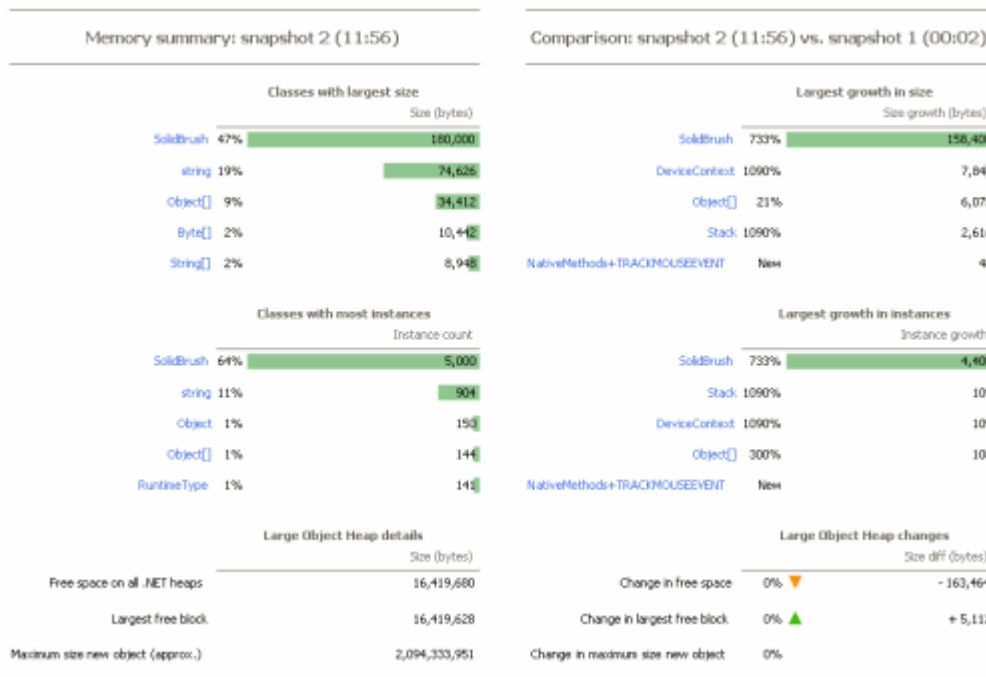


## Memory usage by class

The charts for size and number of instances can be useful starting points for identifying the classes that use most memory.

If you are comparing two snapshots, the charts for growth in size and number of instances show the five classes which are responsible for the largest increase between the two snapshots. This information can be useful for identifying classes which are growing.

Click on a class to show more detail about it on the class list.

---

**Large Object Heap charts**

For an explanation of the Large Object Heap charts, see Finding LOH fragmentation.

# 2. Class analysis

Identify instances of classes which you do not expect to be in memory, or instances with unexpectedly high memory usage. Use object filters to restrict your investigation to objects with specific characteristics or specific parts of the application.

## What to look for in the class list

The **class list** shows detail of memory usage per class. On the summary or class list, look for classes with unexpectedly high memory usage, or a large increase in size between snapshots.

When you enable filters or use the find 🔍 box, the values in the list only include objects that match the selected criteria.

| Namespace | Class Name | Live Size (bytes) | Size Diff (bytes +/-) | Live Instances | Instance Diff (+/-) |
|---|---|---|---|---|---|
| System.Drawing | SolidBrush ✂️ 🔵 | 180,000 | + 150,400 ▲ | 5,000 | + 4,400 ▲ |
| System | string | 74,626 | - 286 ▼ | 904 | - 2 ▼ |
| System | Object[] | 34,412 | + 6,072 ▲ | 144 | + 108 ▲ |
| System | Byte[] | 10,442 | 0 ▬ | 4 | 0 ▬ |
| System | String[] | 8,948 | 0 ▬ | 16 | 0 ▬ |
| System.Windows.Fo... | DeviceContext | 8,568 | + 7,848 ▲ | 119 | + 109 ▲ |
| System.Collections | Hashtable+bucket[] | 8,184 | 0 ▬ | 31 | 0 ▬ |
| System.Configuration | FactoryRecord | 5,320 | 0 ▬ | 95 | 0 ▬ |
| System.Collections | Stack | 2,856 | + 2,616 ▲ | 119 | + 109 ▲ |
| System | RuntimeType | 2,820 | + 20 ▲ | 141 | + 1 ▲ |
| System.Drawing | Point[] | 2,700 | 0 ▬ | 75 | 0 ▬ |
| **ShapePainter** | **EllipseShape** | 2,640 | 0 ▬ | 66 | 0 ▬ |
| System | Int32[] | 2,556 | 0 ▬ | 24 | 0 ▬ |
| **ShapePainter** | **RectangleShape** | 2,360 | 0 ▬ | 59 | 0 ▬ |
| **ShapePainter** | **TriangleShape** | 2,100 | 0 ▬ | 75 | 0 ▬ |
| System | Char[] | 1,910 | 0 ▬ | 23 | 0 ▬ |
| System | Object | 1,800 | - 12 ▼ | 150 | - 1 ▼ |

If you are *performing a general check on memory usage*, or *checking where most memory* is used, it can be useful to start by looking at **Live Size** or **Live Instances**. Click on the column heading to sort the column, and look for classes with unexpectedly large size or number of instances.Right-click on a class and select ⧉ **Show Class Reference Explorer** to see where instances of the class are being referenced.

If you are *looking for a memory leak*, it can be useful to begin by looking for unexpected differences between two snapshots in the **Size Diff** or **Instance Diff** columns. Click on the column heading to sort the column, and then look for classes whose memory usage or instance count has increased significantly. Use the **Comparing snapshots** filters to focus your analysis, depending on the snapshots you are comparing: if you are looking for objects which exist in both the baseline and the current snapshot, select **Only surviving objects**; if you are looking for objects which were created between the two snapshots, select **Only new objects**.

- If the class that looks interesting is one you recognize, right-click and select ![icon]
  **Show Instance List** to look for instances of the class.

- If the class that looks interesting is not one you recognize, right-click and select ![icon]
  **Show Class Reference Explorer** to find out where instances of the class are being referenced.

## Tips

### Namespace and Class Name
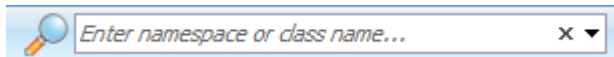


To find a specific namespace or class, type part of the name in the find box. This can be useful, for example, if you are checking back on a memory leak you have fixed. In many cases, we do not recommend starting your investigation by looking for specific classes; instead, start by looking at the size or instances columns.

We recommend this approach because a lot of the code being executed by your application is likely to be part of the .NET framework libraries or other third-party libraries, so you are likely to see leaks in classes which are not your own - even where your code is the cause of the leak.

### Live Size and Live Instances

The **Live Size** column shows the total size of instances of the class in the current snapshot.

The **Live Instances** column shows the total number of instances of the selected class in the current snapshot.

The values do not include instances of classes referenced by the selected class.

These values can be good starting points for finding out where most memory is being used by your application.

To investigate why a class has a large size or high number of instances:

- look at instances of the class on the instance list

or:

- investigate whether another class is keeping instances of your class in memory unexpectedly, on the class reference explorer

### Size Diff and Instance Diff

When you compare two snapshots, the **Size Diff** and **Instance Diff** column show the differences between the baseline and current snapshots.

An unexpected increase in the size of a class or number of instances may indicate a memory leak. For example, if you perform an action where you expect new objects to be

cleaned up between the snapshots (such as opening and closing a dialog box), you would not expect the class to increase in size or number of instances.

There are also cases where an increase in size or number of instances does not indicate a leak:

- For an application that includes a text editor, the size of the text buffer would be expected to increase as the user adds more text to a document. In this case, the **Size Diff** column for the text buffer class shows an increase in size, but this is not an indication of a memory leak.

- For an application with a text editor backed by a DOM, the number of nodes of a DOM would be expected to increase. In this case, the **Instance Diff** column for the DOM classes shows an increase, but this is not an indication of a memory leak.

## What to look for on the class reference explorer

The **class reference explorer** shows reference relationships between classes, which can help identify instances which are unexpectedly kept in memory. This can be particularly useful if your snapshot analysis identifies a class you do not recognize: Use the explorer to follow chains of references to the class, until you reach a class you recognize which may be responsible for the memory usage.

When you apply filters, the class reference explorer only includes the filtered objects.

The class reference explorer has two different views:

- Categorized references view
- All references view

## Categorized references view

This is the default view. In this view, instances of the selected object are categorized by the shortest path to a GC Root. The path responsible for holding most instances of the selected class is shown at the top.

To see the instances that each category contains, click your selected class (in black, on the right), and then click **Show instance list**.

Hover over the image to see an enlarged view.

---

In the all references view, your selected class is shown at the center of the graph (in black). Classes to the left have instances which reference any instance of the selected class; classes to the right have instances which are referenced by at least one instance of the selected class.
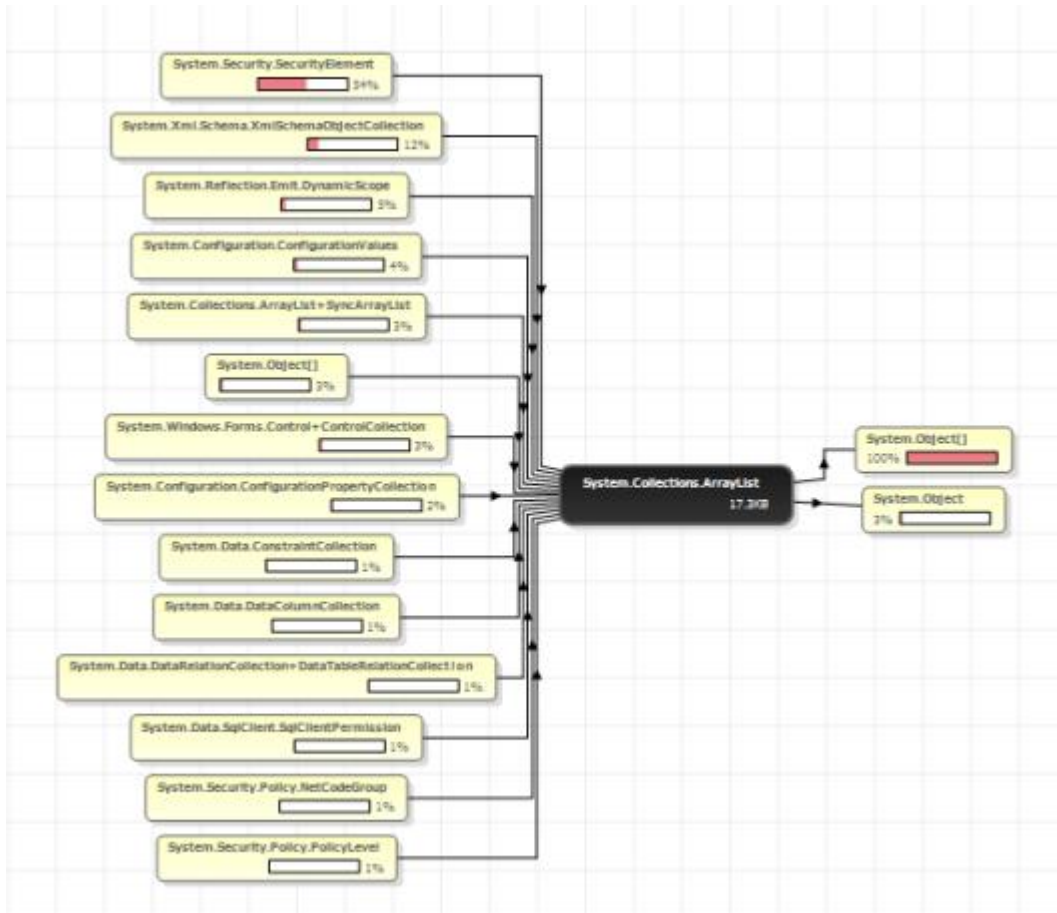


**Tips**

- The percentage shown on a class indicates the proportion of instances of the selected class (in the center) that are connected to it by references along that path. Look at the percentages on classes to the left of the selected class to assess which classes are responsible for instances of the selected class being in memory.

- To find out why your selected class is unexpectedly large, look at the class that is responsible for most references to the selected class.

  ♦ In the categorized references view, this is on the top row, to the left of the selected class. Click the class to expand the graph, showing classes that reference this class.

  ♦ In all references view, this is the class at the top-left of the graph. Follow the path to the left to see classes that reference this class.

- The graph shows all references between classes, so you may find that as you expand classes and follow references along a particular path you start to see the same classes repeatedly in the path. This is a circular reference chain, and you are unlikely to find useful information by continuing to follow it. Instead, click **Show instances of this class on this path** to display the instance list for one of the classes on this path, and then display the object retention graph for an instance, to investigate why the instance is in memory.

- If you are looking for the Class Reference Explorer from ANTS Memory Profiler 6 and earlier, select All references view.

# 3. Instance analysis

Investigate what is keeping a selected object in memory unexpectedly.

## What to look for in the instance list

The **instance list** shows detail for all instances of a class, which can help identify instances which are the likely cause of a memory problem.

When you enable filters, the list only includes objects that match the selected criteria.

| New Object | Value | Size (bytes) | Size with Children (bytes) | GC Root Object | Distance from GC Root |
|---|---|---|---|---|---|
| No | ⊞ Signature ⇕ | 48 | 60 | No | 7 |
| Yes | Signature | 48 | 60 | No | 7 |
| No | Signature | 48 | 68 | No | 3 |
| Yes | Signature | 48 | 72 | No | 10 |
| Yes | Signature | 48 | 68 | No | 10 |
| Yes | Signature | 48 | 72 | No | 10 |
| Yes | Signature | 48 | 68 | No | 10 |
| Yes | Signature | 48 | 72 | No | 10 |
| Yes | Signature | 48 | 68 | No | 10 |
| Yes | Signature | 48 | 72 | No | 10 |

The instance list is useful for understanding what instances of a class are in memory, and for identifying objects which are likely to be involved in a memory leak.

Click ⊞ in the **Value** column to find out more about the properties of a specific instance.



## Tips

### New Object

The **New Object** column indicates whether the object was created between two snapshots you are comparing, or whether it existed already in the earlier snapshot. (This column is only populated when you are comparing two snapshots.)

When you are comparing two snapshots, either *Yes* or *No* in the **New Object** column may be an indication of leaked objects, depending on when you took your snapshots.

- *Yes* may indicate a memory leak when you are comparing snapshots and you expect instances of the selected class to be cleaned up between snapshots. For example, you take a snapshot before and after opening and closing a dialog box. Objects created by the action should be cleaned up before you take the second snapshot, so new objects in the second snapshot are likely to indicate a memory leak.

  When you investigate these new objects further, apply the **Only new objects** filter; this ensures you are only looking at objects which are new in the second snapshot.

- *No* may indicate a memory leak when you are comparing snapshots and you expect instances that exist before the first snapshot to be cleaned up by an action you take between snapshots. For example, you populate a list with data, and then take a snapshot before and after clearing the list. Objects created before the first snapshot should be cleaned up before you take the second snapshot, so old objects in the second snapshot are likely to indicate a memory leak.

  When you investigate these objects further, apply the **Only surviving objects** filter; this ensures you are only looking at objects which exist in both snapshots.

**GC Root Object**

The **GC Root Object** column indicates whether the object is a GC root object. A GC root can be any storage slot to which the running program has access, such as a local variable, static variables, or even a CPU register. (Strictly speaking, the object itself is not the GC root; the storage slot that holds the reference to the object is the GC root.)

When the garbage collector runs it determines which objects are not garbage by walking the heap, starting at the GC roots. Objects which can be reached by following a chain of references from a GC root are designated as not garbage, and are not collected.

To find out why an object in the instance list is a GC root, right-click on it and then select ⚖ **Show object retention graph**. Information on the graph shows why the object is a GC root.
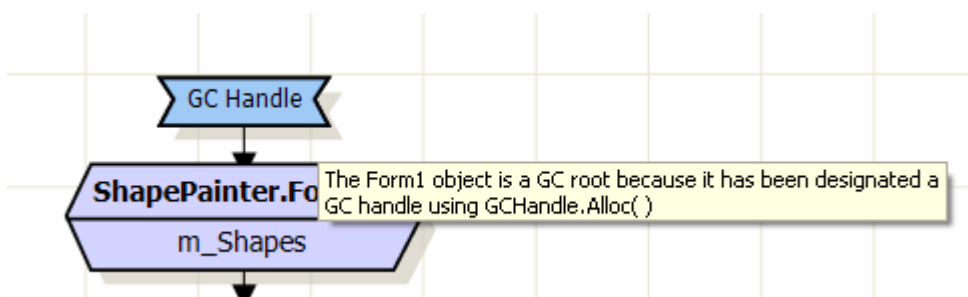


GC root objects are not usually the source of memory leaks. However, they can be useful in *finding* memory leaks because there is always a chain of references between the leaked object and one or more GC roots. To enable the garbage collector to clean up the object, you need to break this chain of references by changing your code to remove one of the "links" in the chain.

The GC Root Object column shows 'Yes - Weakly Referenced' if the object is weakly referenced. Objects with weak references are often used for cacheing because these objects can be destroyed by the garbage collector if memory becomes low. For this reason, weakly referenced objects are not usually the source of memory leaks.

### Distance from GC Root

The **Distance from GC Root** column shows the number of references in the chain between the object and its nearest GC root.

It is likely that shorter, more obvious chains of references between objects and their GC roots have been broken already. Often objects which are at a greater distance from a GC root may be involved in a memory leak, because the chain of references from the GC root to the object is more complex.

### Size with children

The **Size with children** column shows the size of the object and any object that it references that is further away from a GC root. This means that the value is a realistic estimate of the amount of memory that would be saved by removing a particular object from memory.

## What to look for on the object retention graph

The **object retention graph** shows chains of references between GC roots and your selected object. Start at your selected object and follow the chains of references up towards the GC roots, to identify references that are preventing the garbage collector from collecting your object. When you find an unexpected reference, modify your code to break the reference, and then profile the application again to check the problem is fixed.
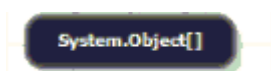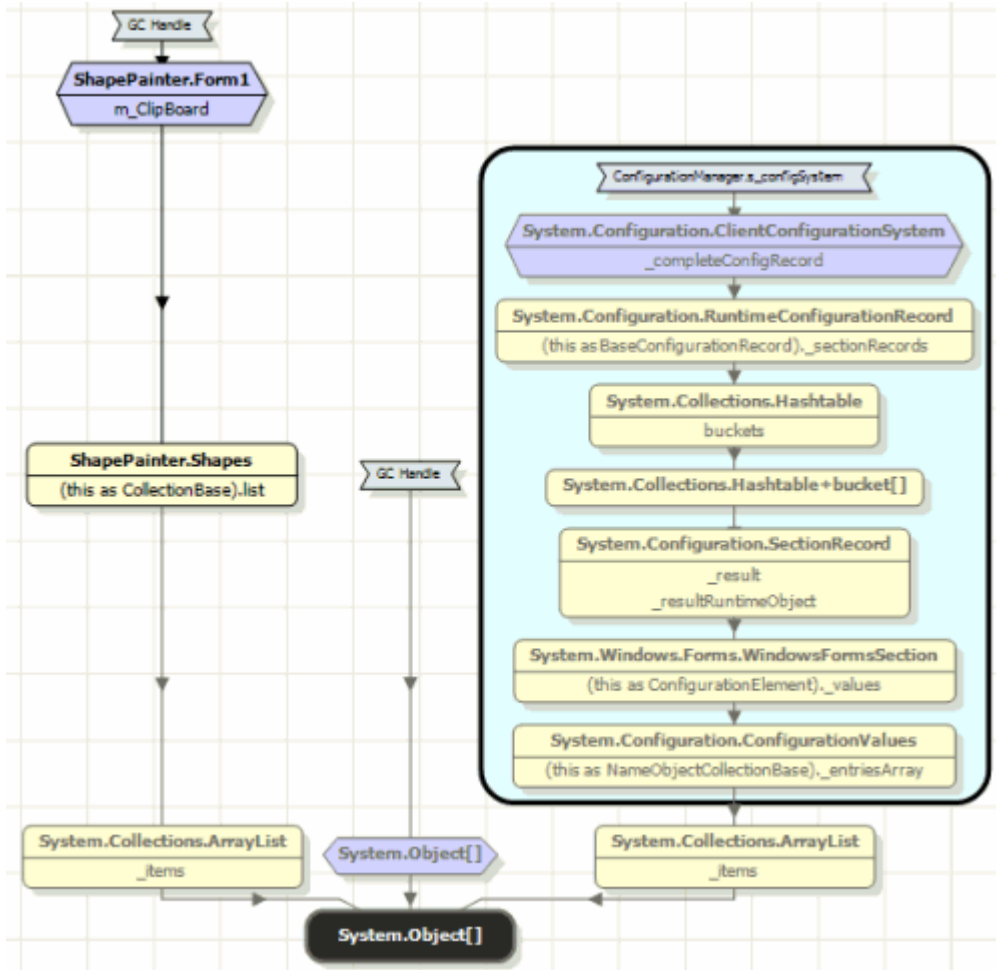



The object you selected.


GC root object.


Reason why the object below is a GC root.
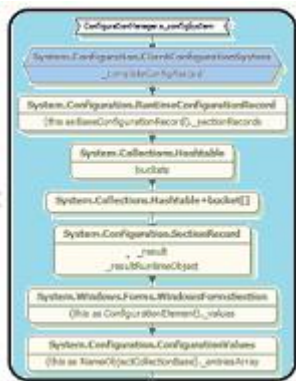
Group of strongly-connected objects (see tips below).
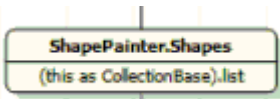


Object for which you have source code.
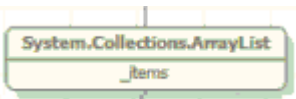


Object for which you do not have source code.
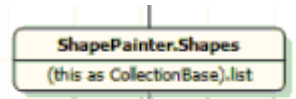


The simplest path between these two objects.
(Note that if you break this link, the objects may still be connected by a more complex link.)
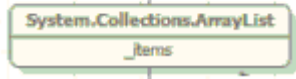
For more information about specific objects on the graph, move the mouse cursor over the object; details are displayed in a tooltip.

**Tips**

- The object retention graph only shows the shortest chain of references from each GC root to your selected object. When you break this chain of references, the object may still be kept in memory by a longer chain of references.

  After you have modified your code to break the first chain of references, profile your application again; the object retention graph updates to show the chain of references which is now the shortest chain. You will need to modify your code again, to break this chain of references, and repeat until all the chains of references are broken and the object is no longer in memory.

- Objects grouped in a box are strongly connected; every object references every other object in the group (the reference may be direct or indirect). To remove an object from memory, you do not normally need to break all the references between a GC root and your object: only *one* of the references needs to be removed to prevent your object from being kept in memory.

  However, the relationship between strongly connected objects is complex, so in this case you may need to break more than one reference to prevent your object from

---

being held in memory. Break the references one at a time, and take new snapshots each time to check whether your object is still in memory.

- If there is an event handler in the chain of references from a GC root to your object, look at the objects that directly reference the event handler; these references are often a good point to break the chain of references to your object.

- If your graph shows an object which seems to not be referenced by anything, it may be because this object is on the finalizer queue. The graph hides finalizer queue GC roots by default, because they do not normally indicate a memory problem. To show these objects on the graph, clear the **Hide finalizer queue GC roots** option (in the bar above the graph).

  If the graph shows your object is being kept in memory by an object on the finalizer queue, take another memory snapshot. Taking a snapshot forces the garbage collector to run, so your object should now be removed from memory.

# Using filters to find objects

Use the object filters to focus your memory usage investigation on objects that are more likely to be of interest.

Select one or more of the filters to show only objects that meet all of the selected criteria. Other objects are hidden.

The bar above the results area indicates the number of objects affected by the current filters:
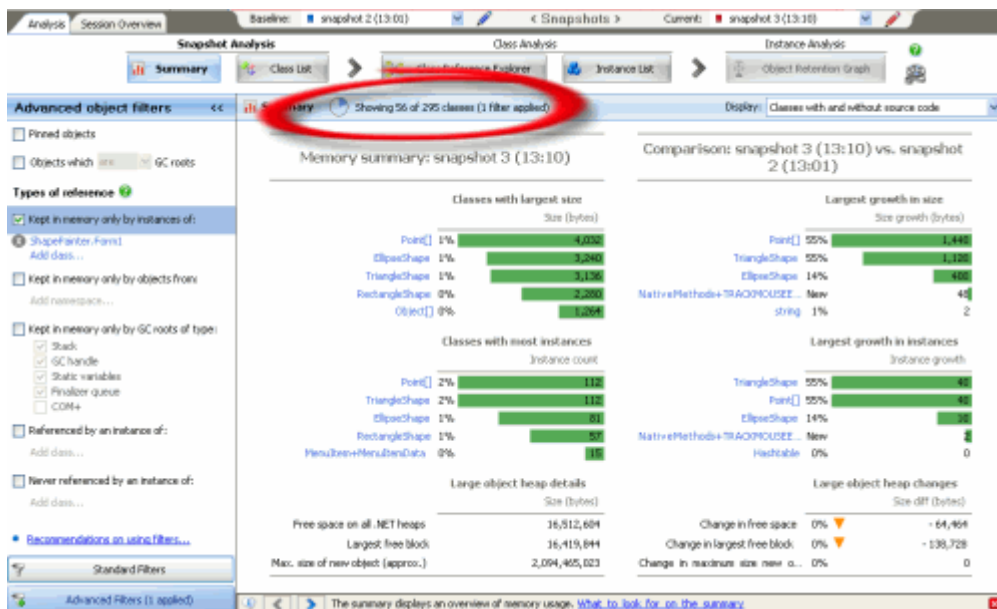


When using filters, you may see some objects which do not match the filter criteria. If this happens, take another snapshot. This forces a garbage collection so that any objects on the finalizer queue are removed.

## Suggestions for using filters

Filters alone cannot identify memory problems, but they can help narrow down your search space. The following notes suggest filters that can be useful in some conditions.

### Filters which can help find a memory leak

The following filters can be useful in finding a memory leak:

**Disposed objects which are still in memory**

**Kept in memory only by event handlers**

**Kept in memory only by disposed objects**

**Only zombie objects** (in the **Comparing snapshots** filters)

Note that the 'Disposed objects which are still in memory' and 'Kept in memory only by event handlers' filters are not available when profiling a .NET 1.1 application.

### Filters which can help focus on a specific part of your application

The following filters can be particularly useful when you want to focus on a specific part of an application, or exclude a specific part of an application:

> **Objects on** *large object / Gen 0 / Gen 1 / Gen 2* **heap**
>
> **Kept in memory only by objects from** *namespace* (in the **Advanced filters**)
>
> **Never referenced by an instance of** *class/interface* (in the **Advanced filters**)

Note that the 'Objects on large object / Gen 0 / Gen 1 / Gen 2 heap' filter is not available when profiling a .NET 1.1 application.

You can also focus on a specific part of an application by using the **Process** selection filter.

See Strategies for memory profiling for examples of how to incorporate filtering in your memory profiling workflow.

### Relationships between filters

When you apply multiple filters, only objects which match **all** the filters are shown (i.e. there is an AND relationship between filters).

For example, select the following filters:

- **Disposed objects which are still in memory**
- **Objects on** *Gen 1* **heap**

Objects are only shown if Dispose() has been called on them *and* they are on the Gen 1 heap.

### Relationships between objects

Some of the filters enable you to narrow down your search for memory problems by concentrating on certain types of relationships between objects.

### "Referenced by"

Objects may be in memory because another object references them; the object is on *at least one of* the chains of references between the selected object and a GC root.

---

Example: objects referenced by *Object 1*
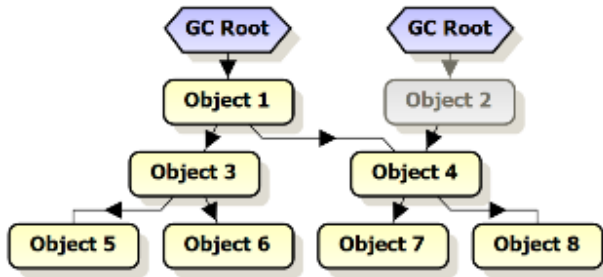


In this example all objects except *Object 2* are referenced by *Object 1*, either directly or indirectly. Note that the filter selection includes the specified object, *Object 1*.

## "Kept in memory only by"

Some filters show objects where the selected object is in **all** chains of references between the objects and a GC root - that is, objects are kept in memory *only* by the selected object.

Example: objects kept in memory by *Object 1* (note that the filter includes the selected object, *Object* 1)
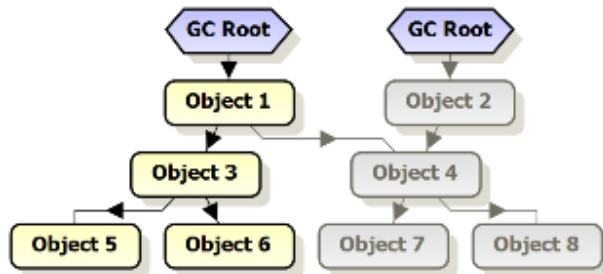


In this example, only four objects are kept in memory only by *Object 1*. *Object 1* is not in all the chains of references between the remaining objects and their GC roots; for example, *Object 4* has another GC root which references it via *Object 2*. Note that the filter selection includes the specified object, *Object 1*.

## See also

# List of object filters

Object filters enable you to focus your analysis on specific types of object or specific parts of the application. This topic describes the available filters, and suggests situations in which they might be useful.

## Standard filters

### Disposed objects which are still in memory

Show only objects on which Dispose() has been called, but which cannot be garbage-collected because a reference to the object still exists in memory.

Disposed objects should not normally be kept in memory, so this filter can be a good indicator of a memory leak.

When you have identified a disposed object with this filter, display it on the object retention graph, then follow the chains of references to identify the objects keeping the disposed object in memory.

This filter is not available when you are profiling a .NET 1.1 application.

### Objects on the *large object / Gen 0 / Gen 1 / Gen 2* heap

Show only objects in the selected area of memory.

This filter is not available when you are profiling a .NET 1.1 application.

### Objects on finalizer queue but not disposed

Show only objects on the finalizer queue on which Dispose() should be called.

### Kept in memory only by event handlers

Show only objects where all chains of references between the object and a GC root go through an event handler.

Examples: objects kept in memory only by event handlers

(note that the event handler is included in the objects shown)
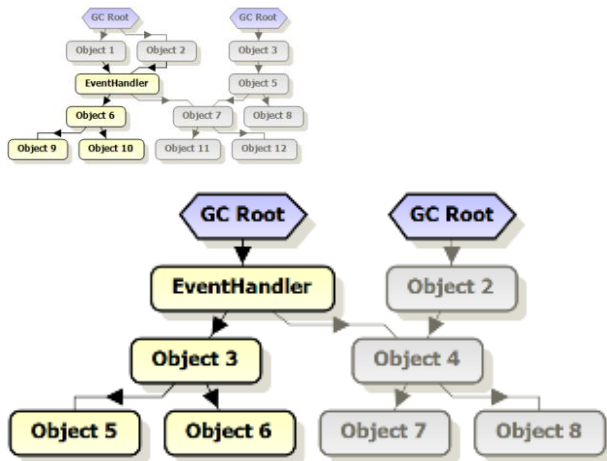


Event handlers should not normally be the only reason an object is kept in memory, so this filter can help identify a memory leak.

When you have identified an object that is kept in memory only by an event handler, display it on the object retention graph, and follow the chain of references from the object to the event handler; it is likely that the cause of the problem is an object close to the event handler.

**Note:** Objects which match your criteria but are on the finalizer queue are *not* shown.

### Kept in memory only by disposed objects

Show:

* disposed objects

*and*

* objects where all chains of references between the object and a GC root go through a disposed object

Examples: objects kept in memory only by disposed objects

(note that the disposed object is included in the objects shown)
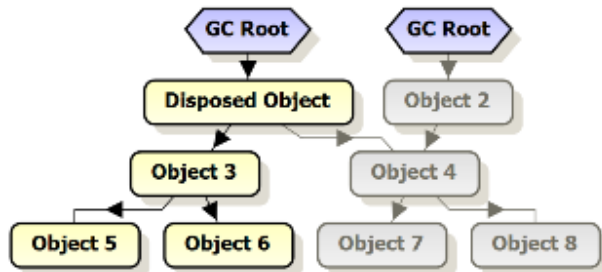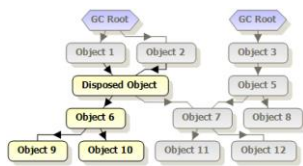




Disposed objects should not normally be kept in memory, so this filter can help identify a memory leak.

When you have identified an object that is kept in memory only by a disposed object, display it on the object retention graph, and follow the chains of references to identify the objects keeping the selected object in memory.

**Note:** Objects which match your criteria but are on the finalizer queue are *not* shown.

This filter is not available when you are profiling a .NET 1.1 application.


Comparing snapshots

Show objects in your current snapshot, based on the comparison with your baseline snapshot.

*Only new objects*
Show only objects created between the baseline snapshot and the current snapshot.

*Only surviving objects*
Show only objects that remain in memory in both the baseline snapshot and the current snapshot.

*Only zombie objects*
Show only objects that showed indications in the baseline snapshot that they would not survive in the current snapshot, but which are still in memory in the current snapshot.

This includes objects with the following characteristics:

- Objects on which Dispose() has been called in the baseline snapshot, but which are still in memory in the current snapshot because garbage collection was prevented for some reason
- Objects on the finalizer queue in the baseline snapshot (this includes objects still on the finalizer queue in the current snapshot and objects which are no longer on the finalizer queue in the current snapshot)

These objects should not normally still be in memory in the current snapshot, so they can be useful for identifying a memory problem.

## Advanced filters

### Pinned objects

Show only objects marked to not be moved in memory. An increasing number of these objects could indicate that the objects are not being unpinned for some reason.

Large numbers of pinned objects can cause performance problems because of the way the garbage collector handles these objects.

### Objects which *are / are not* GC roots

Hide GC root objects or show only GC root objects.

If you are investigating a memory leak, it may be useful to hide GC root objects, because they are not generally the source of leaks.

If you are performing a general check on memory usage, it may be useful to show only GC root objects. Show the class list to see details of classes with instances which are GC roots, and then use the class reference explorer to investigate the relationships between these classes and other classes with instances in memory.

**Tip:** To look for specific types of GC roots, combine this filter with the **Kept in memory only by GC roots of type** filter.

### Kept in memory only by instances of *class/interface*

Show:

- instances of the specified class or interface (including derived types)

*and*

- objects where instances of the specified class or interface exist in all chains of references between the object and a GC root

Example: objects kept in memory only by the class that *Object 4* is an instance of

(note that *Object 4* - which is an instance of the specified class - is included in the objects shown)
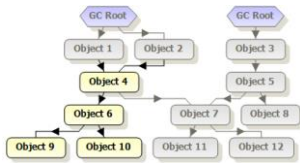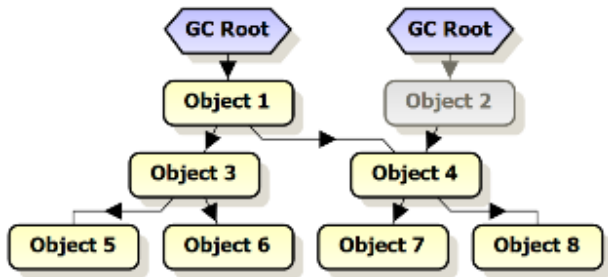


Example: objects kept in memory only by the class that *Object 1* is an instance of
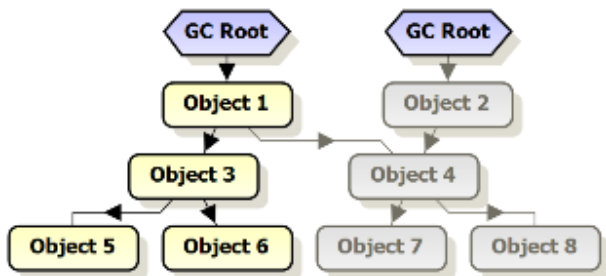


This filter shows objects where instances of the specified class or interface (including derived types) exist in *all* chains of references between the object and a GC root. To show objects where instances of the specified class exist in *at least one* chain of references (but not necessarily in *all* chains of references) between the object and a GC root, use the **Referenced by instances of *class/interface*** filter.

**Note:** Objects which match your criteria but are on the finalizer queue are *not* shown.

**Kept in memory only by instances of *namespace***

Show:

- objects from the specified namespace

*and*

- objects where instances of classes from the specified namespace exist in all chains of references between the object and a GC root

Example: objects kept in memory only by a class from the namespace that *Object 4* is an instance of

(note that *Object 4* - which is an instance of the a class in the specified namespace - is included in the objects shown)
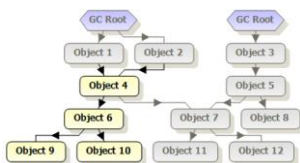
Example: object kept in memory only by a class from the namespace that *Object 1* is an instance of
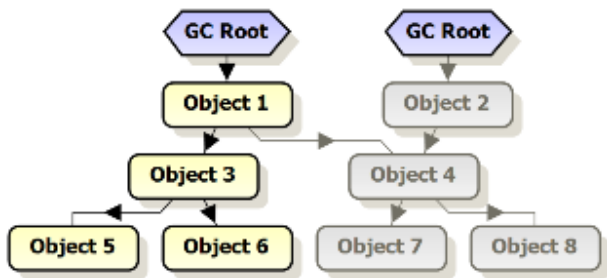


When you select multiple namespaces for this filter, objects are shown if they are instances of classes in any of the namespaces.

**Note:** Objects which match your criteria but are on the finalizer queue are *not* shown.


**Kept in memory only by GC roots of type *GC root type***

Show objects with GC roots of the specified types (includes objects which are GC roots).

When you select more than one type of GC root, objects are shown if they have a GC root which is any of the selected types.

**Tip:** To look for specific types of GC roots, combine this filter with the **Objects which *are* GC roots** filter.


**Referenced by an instance of *class/interface***

Show only objects which are referenced by one or more instances of the specified class or interface (including derived types). The reference may be direct or indirect.

Example: objects referenced by the class that *Object 4* is an instance of

(note that *Object 4* - which is an instance of the specified class - is included in the objects shown)
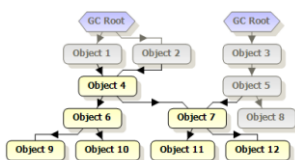
Example: objects referenced by the class that *Object 1* is an instance of



When you select multiple classes or interfaces for this filter, objects are only shown if they are referenced by all of the selected classes and interfaces.

This filter shows objects where instances of the specified class exist in *at least one* chain of references between the object and a GC root. To show objects where instances of the specified class exist in *all* chains of references between the object and a GC root, use the **Kept in memory only by instances of *class/interface*** filter.

**Never referenced by an instance of *class/interface***

Show only objects which are never referenced by an instance of the specified class or interface (including derived types). This includes direct and indirect references.

Example: objects never referenced by the class that *Object 4* is an instance of
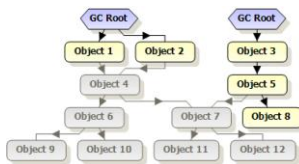


When you select multiple classes or interfaces for this filter, objects are only shown if they are not referenced by any of the selected classes or interfaces.

**Note:** Objects which match your criteria but are on the finalizer queue are *not* shown.

Some examples of when you might want to use this filter:

- You know that all the data in your application should be referenced by a single main class. Apply the **Never referenced by an instance of *class/interface*** filter to remove objects referenced by the main class from the results.

- Your application performs caching (for example, web applications that cache the query results). Cached objects are deliberately kept in memory for a period of time. Apply the **Never referenced by an instance of *class/interface*** filter to exclude objects referenced by a cache class (for example, *System.Web.Caching.Cache*, for ASP.NET applications).

# Video tutorials

Watch the video tutorials to see examples of ANTS Memory Profiler in action, and learn more about some areas of functionality:

- Finding a memory leak (http://www.red-gate.com/products/ants_memory_profiler/overview.htm)
  One of the Support Engineers for .NET tools takes you through the new ANTS Memory Profiler and illustrates how to use the tool to find a memory leak.

- Using filters to speed up your search for memory problems (http://www.red-gate.com/products/ants_memory_profiler/filters.htm)
  Stephen Chambers, the Usability Engineer who worked on this profiler, demonstrates how you can use the filters to speed your search for a memory leak.

- Using the class reference explorer to find a memory leak (http://www.red-gate.com/products/ants_memory_profiler/class_reference.htm)
  Stephen Chambers, the Usability Engineer who worked on this profiler, demonstrates how you can use the Class Reference Explorer graph to locate your memory leak.

- Using the timeline (http://www.red-gate.com/products/ants_memory_profiler/timeline.htm)
  Stephen Chambers, the Usability Engineer who worked on this profiler, shows you how to use the timeline in the new ANTS Memory Profiler 5.

For a general introduction to .NET memory management:

- .NET memory management (http://www.red-gate.com/products/ants_memory_profiler/DOTNET_Memory_Management/index.html)

# Acknowledgements