Finding and fixing a memory leak

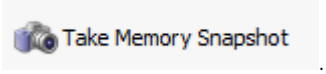
Before you look for a memory leak, we recommend that you first check for large object heap fragmentation and use of unmanaged memory.

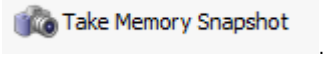
If you are not familiar with memory profiling, you might find it helpful to read about the .NET heaps before you start.

Your approach to finding the source of the leak will depend on whether you have a hypothesis as to what is causing the problem.

If you think you know what is causing the problem

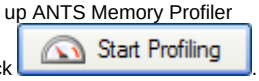
You may already have an idea of the action that might be causing the problem. In this case, use ANTS Memory Profiler to avoid time-consuming trial-and-error investigations in the code.

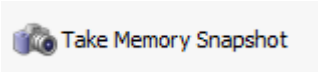
1. Start ANTS Memory Profiler and start profiling your application.
2. Use your application until the point where you are interested in its memory.
3. Click 

.
4. Cause the action or method to take place (for example, open the dialog box).
5. Click 

.
6. Switch to the **Class List** and try each of the following approaches to check whether the type that you suspect is the source of the problem:
    a. Type the name of the type which you suspect is causing the problem into the search box.
    b. On the filter panel, click **Filter by Reference**. Select **Kept in memory exclusively by**, click **Add class/interface**, and enter the name of the type there.
    c. If you know that the class should only be in memory because it is referenced only by one other class, check using the **Never referenced by** filter.
7. If this procedure does not reveal any problems, continue reading 'Analyzing heap usage', below.

If you do not know what could be causing the problem

1. Set up ANTS Memory Profiler
2. Click 

.
3. When you see the **Bytes in all heaps** graph on the timeline increase substantially, click 

twice, a few seconds apart.
4. Continue reading at 'Analyzing heap usage' below.

Analyzing heap usage

Our best advice for analyzing heap usage is to ensure that you perform the analysis methodically, noting the results so that you can check that any changes made to your code have fixed the problem.

For each suspicious class (see the list below), in turn:

1. Show the **Instance Categorizer** graph.
2. If the class is one of yours, switch to the **All references** mode, so you can see the objects this class references.
3. Check how the class is being kept in memory by looking at classes displayed to the left of your selected class. Check especially for any event handlers referencing your selected class.
4. If a path looks incorrect, switch to the **Instance Retention Graph** to show how an instance is referenced along that path.

Suspicious classes are:

- The largest classes (see the Summary or Class List)
- Classes displayed when the **Kept in memory exclusively by disposed objects** / **by event handlers** **filters** are applied. Having objects in the latter is an especially good indicator of a leak (or poor coding practice).
- Classes which ought not to be displayed when the **Survivors in growing classes** and/or **New objects** filters are applied.

> For the largest classes, you can jump straight to the categorized references graph from the summary. You do not need to generate the Class List every time.

Solving the memory leak

Once you have identified a path which is incorrect:

1. Break the incorrect references in your code
2. Retest the leak.