JetBrains dotTrace 3.1 Reviewer's Guide

Welcome!

On behalf of JetBrains and the entire dotTrace development team, we want to personally thank you for taking the initiative to review **dotTrace Profiler 3.1** – our comprehensive and easy-to-use profiling solution for the .NET platform.

dotTrace Profiler is standalone software that offers performance and memory profiling for .NET applications, including desktop applications, ASP.NET applications running on IIS, and Windows services.

Here is what dotTrace Profiler brings to the table:

1. Fastest in-class profiling tool in the market
2. Powerful filtering, searching, navigation and analysis capabilities
3. Tight integration with Microsoft Visual Studio
4. Automatic comparison of performance snapshots
5. Multiple profiling modes: 4 for performance and 3 for memory profiling
6. Unlimited number of snapshots can be opened and viewed at the same time
7. All features are instantly available from the keyboard (no switching between keyboard and mouse)
8. Multiple views to examine your application from different points
9. Command line support allows easily integrating dotTrace into your daily build process
10. dotTrace API lets you control and utilize dotTrace's powerful features from your own applications

System Requirements

Before installing dotTrace, please make sure that the following requirements are met:

- A processor with IA32 or X64 architecture
- Microsoft Windows 2000, XP, 2003 Server or Vista
- Microsoft .NET Framework 1.1 or 2.0
- Internet Explorer 6 or later
- 512 megabytes of RAM or more is recommended
- 40 megabytes of hard drive space
- Please note: memory profiling works only with Microsoft .NET Framework 2.0 applications

Installing dotTrace

You'll find the installer at http://www.jetbrains.com/profiler/download/index.html.
Choose your system architecture, download the corresponding installation package, and run it to start the Installation Wizard. Follow the instructions provided.

Getting Started

Using dotTrace is straightforward and easy. A typical profiling session consists of the following:

- Choose the application you want to profile from dotTrace (on the Welcome screen, click **Profile Application**).
- Select performance profiling or memory profiling as needed.
- Let your application run, and/or perform specific actions in it to target certain functionality.
- Tell dotTrace when to fetch collected data (in the Control Profiling dialog, click **Get Snapshot** if profiling performance or **Dump Memory** if profiling memory).
- Review and analyze the collected data.
- Save the snapshot (*Ctrl+S*) or export it to a file (**View → Export Current View**) for later use or reference.
- Capture additional snapshots and/or run additional sessions as needed.

Please see below for step-by-step walkthroughs of the typical Performance and Memory Profiling Scenarios:

Performance Profiling

Memory profiling:

- Dump memory mode
- Memory difference mode

Typical Profiling Scenarios — Step-by-step Walkthroughs

Performance Profiling

Step 1. Launch demo application from dotTrace

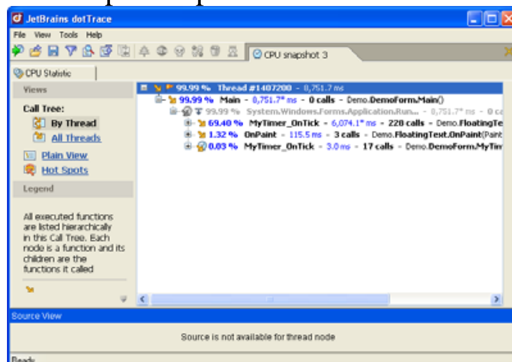
1. Start dotTrace.
2. On the Welcome screen, click the "Profile sample application" link. This launches the sample application (comes bundled with dotTrace). The Control Profiling dialog also opens:

It lets you start and stop the profiling of your application and capture snapshots as you go along.

Step 2. Profile it

1. Switch to the sample application, which runs in the background, and observe it for a few seconds. Switch to the Rainbow mode (slower):

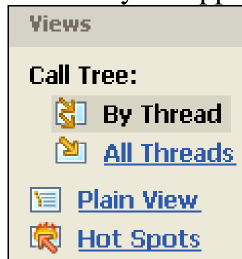
2. Click the **Start Profiling** button in the Control Profiling dialog. Data are now being collected on our application.
3. After a short while (10-20 seconds is enough), click the **Get Snapshot** button in the Control Profiling dialog. Data collection is stopped, and the collected data are processed into a *performance snapshot*.
4. The snapshot opens:



You can close the Control Profiling dialog for now.

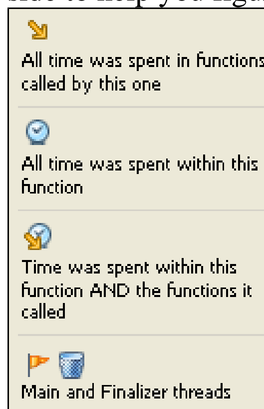
Step 3. Analyze performance snapshot

1. By default, the Call Tree is shown. You can switch views in the top left-hand corner, to examine your application from different points:



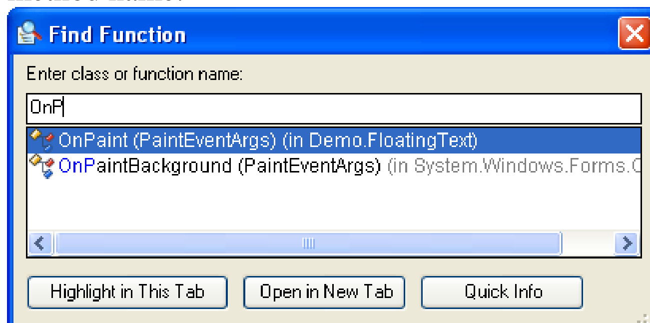
But let's stay with the Call Tree for now.

2. Play with the Call Tree it to see what it has to offer. There is a legend on the left-hand side to help you figure out what all the icons mean:

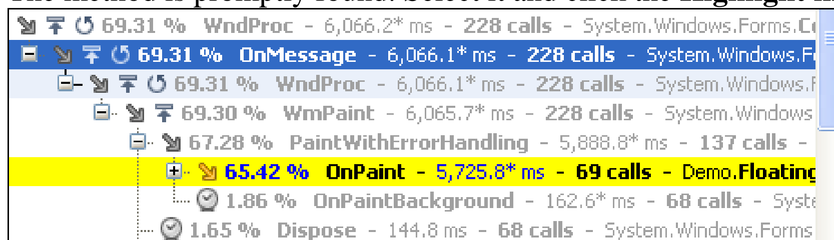


Step 4. Locate performance bottleneck

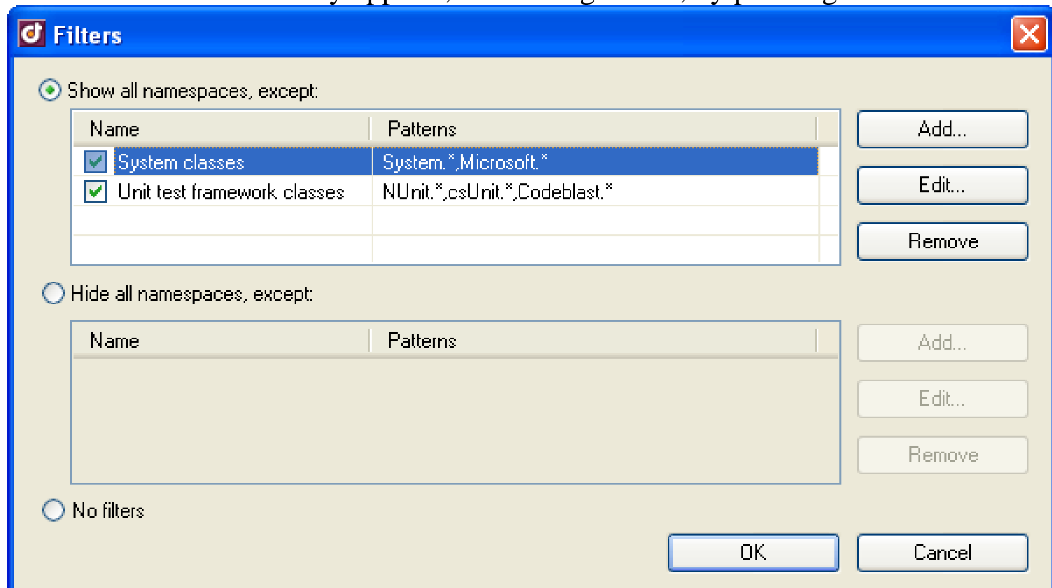
1. The standard event handler for painting in this type of applications is called **OnPaint**. The quickest way is probably to find it by name. Press *Ctrl+F* and start typing the method name:



2. The method is promptly found. Select it and click the **Highlight in This Tab** button:

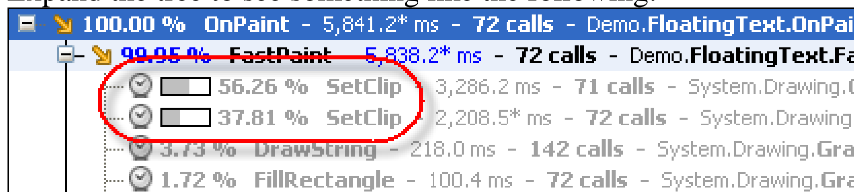


3. As you can see, a lot of calls are filtered to help you better see the call structure. You can see the filters currently applied, and manage them, by pressing *Ctrl+Alt+F*:



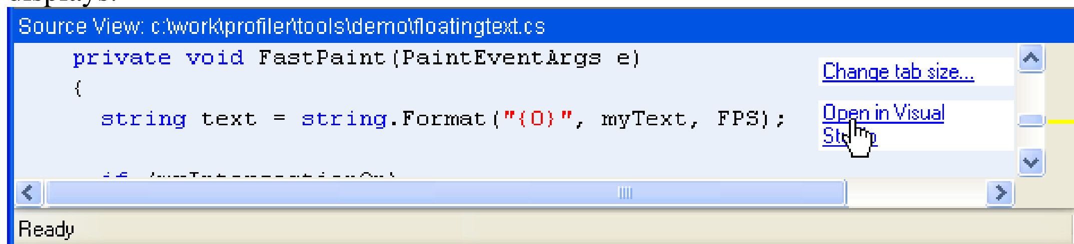
4. You can hide even more system calls (which are grayed out) by clicking **View → Filters → Fold filtered calls by default**.
5. Let's inspect the **OnPaint** method in more detail. Select the node and press *Ctrl+T*. The method is opened in its own tab:

6. Expand the tree to see something like the following:



You can see that it's the SetClip functions that consume over 90% of painting time. They are called from the user-made **FastPaint** method. This is where you should focus your code optimization efforts.

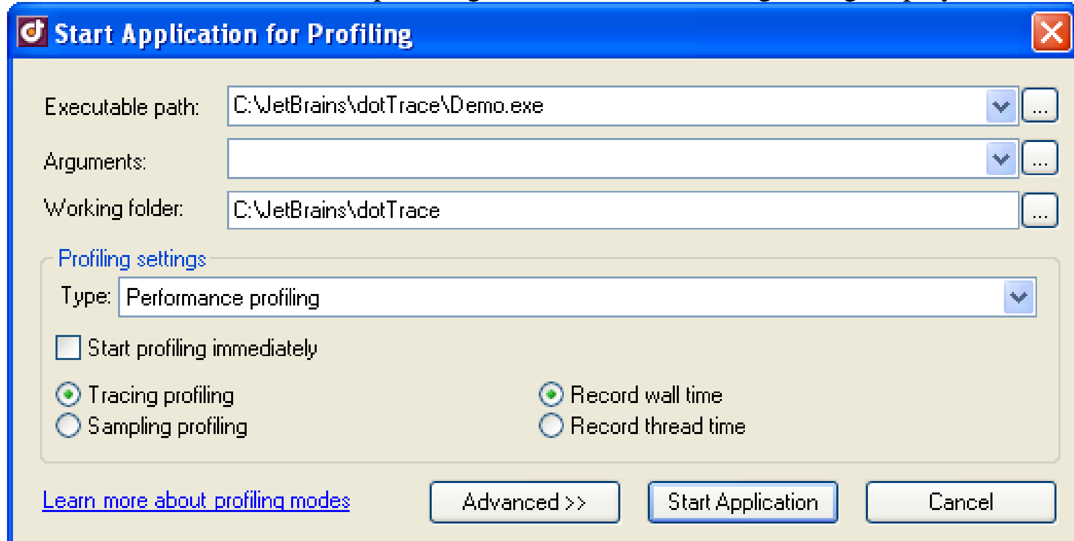
7. To see the code, use the Source View pane, located at the bottom of the snapshot screen. Click the "browse" link to locate the .pdb file. Then the underlying source code displays:



Also, when the corresponding solution is open in Microsoft Visual Studio, you can click the "Open in Visual Studio" link to navigate to the code. If you edit the code, you will be able to start profiling your new application right from Visual Studio, too!

Step 5. Capture second snapshot and compare it with first one

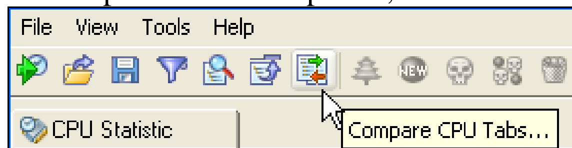
1. Another powerful dotTrace feature is opening *multiple snapshots*. What is more, there is an automated *snapshot comparison tool* to help you detect differences between any two snapshots. Let's see how this is accomplished.
2. Press *Ctrl+A* to start another profiling session. The following dialog displays:



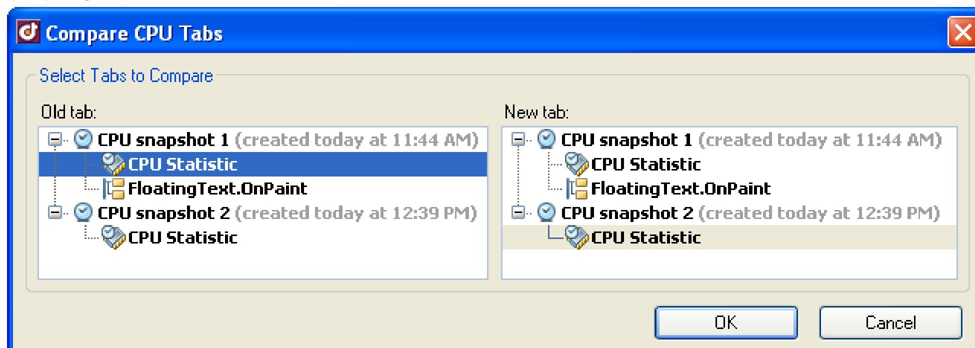
Specify the application executable path and select the options as shown above, then click the **Start Application** button.

3. The application starts. Keeping the application in the faster drawing mode, click the **Start Profiling** button.
4. After roughly the same amount of time as in the first snapshot, click the **Get Snapshot** button. The second snapshot is opened alongside the first one:

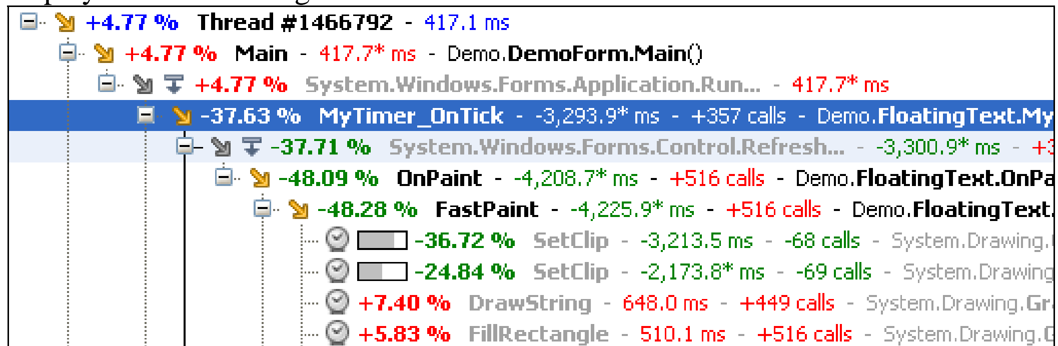
5. To compare the two snapshots, click the following button on the toolbar:



6. On the next screen, you designate which of the open tabs in each snapshot will serve as the "old" and the "new" tabs in the comparison. Select "CPU Statistic" for each and click OK:



7. A comparison snapshot opens. It can be used just the same way as regular ones. It displays function timings and function call counts as so:



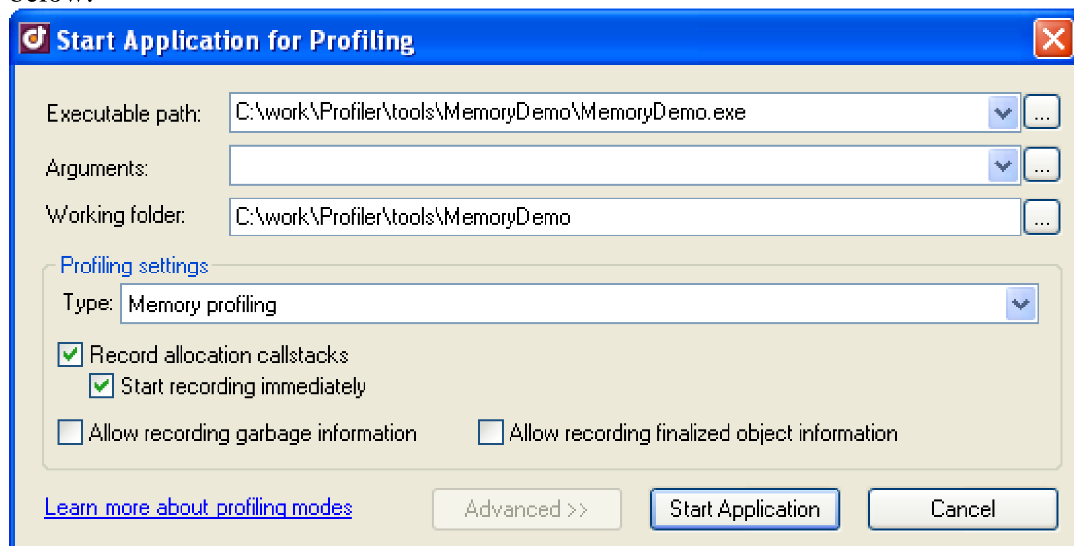
8. Changes are marked red and green. Green figures represent decreases, which likely indicate improvements in performance. Increases are shown in red.
9. So, what do we get from this comparison? In roughly the same total CPU time, the OnPaint method consumes significantly less time than before.

Memory Profiling: Dump Memory Mode

About this profiling mode
The Dump Memory mode allows you to check what objects are currently loaded in memory, which functions allocated them, and more.

Step 1. Launch MemoryDemo application from dotTrace

1. Download the MemoryDemo application package from http://www.jetbrains.com/profiler/documentation/memorydemo.zip. Unzip the file to your computer.
2. Start dotTrace.
3. On the Welcome screen, click the "Profile application" link. In the following dialog, enter the MemoryDemo application execution path and select the options as shown below:

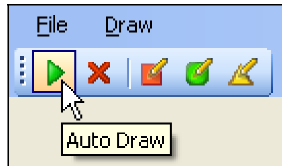


4. Click the **Start Application** button. The MemoryDemo application starts, and the Control Profiling dialog opens:

It lets you start and stop the profiling of your application and capture snapshots as you go along.

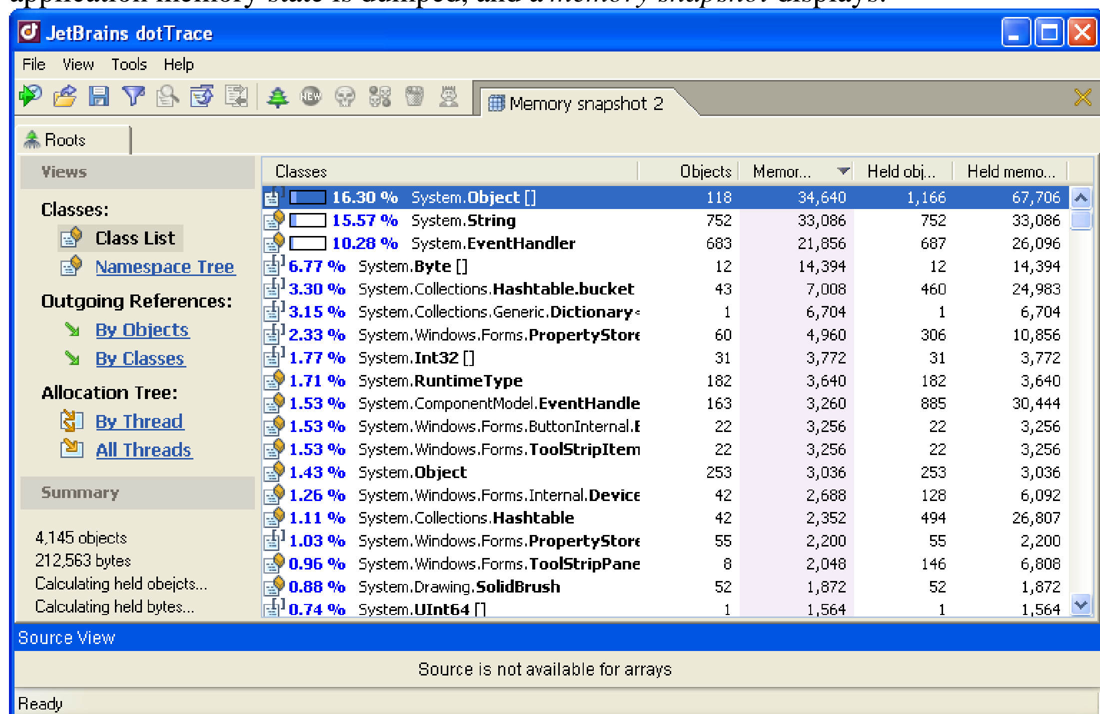
Step 2. Profile it

1. This demo application lets you draw simple shapes such as triangles, rectangles, and ellipses. We will draw a few of each type and then *dump* the current memory state to see what objects remain in memory.
2. Click the Auto Draw button:

   The application will draw a few random shapes.
3. After a few items of each type are drawn, click the button again to stop the drawing.
4. Click the Draw Triangle button:

   Enter a number, for example, 10, and the application will draw 10 random triangles.
5. In the Control Profiling dialog, click the **Dump Memory** button. The current application memory state is dumped, and a *memory snapshot* displays:

   NOTE: Do NOT close the demo application just yet.

Step 3. Analyze memory snapshot

1. The Class List, which is the default view, shows all classes of objects currently allocated in the application's memory. For each class, the following information is provided in four columns:

Classes	Objects	Memory, bytes	Held objects	Held memory, bytes
16.30 % System.Object []	118	34,640	1,166	67,706
15.57 % System.String	752	33,086	752	33,086
10.28 % System.EventHandler	683	21,856	687	26,096

- **Objects** – all objects (instances of a class)
- **Memory** – total memory consumed by all objects of a class
- **Held objects** – all held objects of a class, i.e. those that would be deleted from memory if the class were to be deleted
- **Held memory** – total held memory of a class

2. On the left-hand side, you can find a snapshot summary and a legend to help you figure out the notation.
3. Also on the left-hand side of the screen, let's switch to another view, the Namespace Tree:

4. In this view, classes are grouped by namespace, and this makes it easier for you to find particular classes. In our case, let's look at the classes responsible for drawing. Expand the MemoryDemo namespace node:

Roots

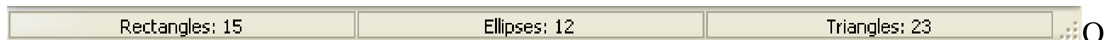
Views

Classes:
Class List
Namespace Tree

Outgoing References:
By Objects
By Classes

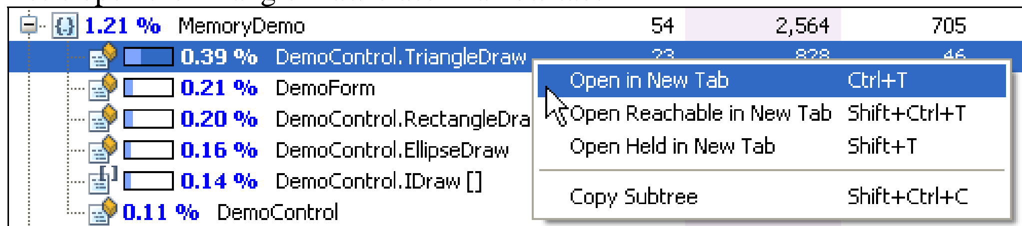
Namespaces and Classes	Objects	Memor...	Held obj...	Held memo...
98.20 % System	4,043	208,735	4,125	211,375
1.21 % MemoryDemo	54	2,564	705	40,570
0.39 % DemoControl.TriangleDraw	23	828	46	1,656
0.21 % DemoForm	1	456	586	35,914
0.20 % DemoControl.RectangleDraw	15	420	30	960
0.16 % DemoControl.EllipseDraw	12	336	24	768
0.14 % DemoControl.IDraw []	2	288	102	3,672
0.11 % DemoControl	1	236	118	4,640
0.59 % Microsoft	48	1,264	50	1,328

5. Here you can see the DemoControl.EllipseDraw, DemoControl.TriangleDraw, and DemoControl.RectangleDraw classes.
   Their object counts should correspond to the ones you see in the MemoryDemo application, which should still be open. Look at the bottom status line of the app to see its counts:

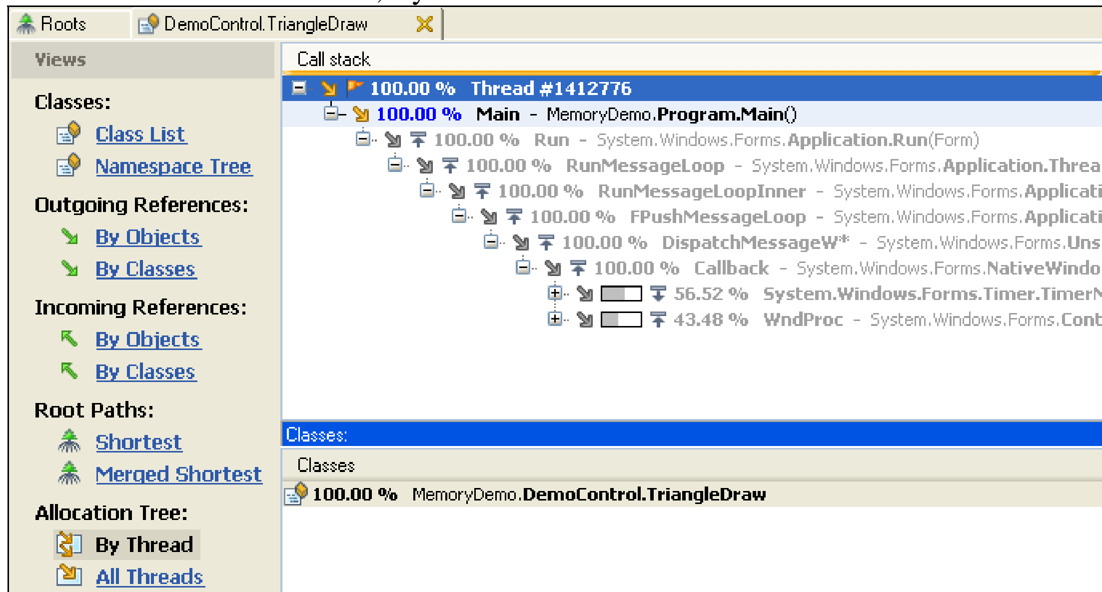


O

   Of course, the exact numbers will differ in your particular run, but they should still perfectly match the ones reported in the memory snapshot.
6. Any subsystem of your application can be opened in its own tab for closer inspection. Let's open the TriangleDraw class in a new tab:



7. This allows us to become familiar with the rest of the views, which can also be very helpful. Click the following views to see what they offer:
   - **Outgoing references, By Classes** – shows all objects references by DemoControl.TriangleDraw
   - **Incoming references, By Classes** – shows all objects that referenced objects of the TriangleDraw class, grouped by class
   - **Allocation Tree, By Thread** (available only when the "record allocation callstacks" option is selected when you start the application for profiling) – shows the Call Stack with the allocations tree
8. Switch to the Allocation Tree, By Thread. It should look as follows:



   This shows exactly which functions allocated the instances of the TriangleDraw class (or any other class, if we focus on it).
9. Now, scroll to the right for allocation statistics. It is easy to make sure that 13 triangle objects were allocated by the timer function (the ones created in the auto-draw mode), while 10 were allocated by the Control class (the ones created manually).

	0	0	Objects
ad #1412776	0	0	23
n - MemoryDemo.Program.Main()	0	0	23
% Run - System.Windows.Forms.Application.Run(Form)	0	0	23
00 % RunMessageLoop - System.Windows.Forms.Application.ThreadContext.RunMessageLoop(Int32, Ap	0	0	23
00.00 % RunMessageLoopInner - System.Windows.Forms.Application.ThreadContext.RunMessageLoop	0	0	23
100.00 % FPushMessageLoop - System.Windows.Forms.Application.ComponentManager.FPushMessa	0	0	23
100.00 % DispatchMessageW* - System.Windows.Forms.UnsafeNativeMethods.DispatchMessage	0	0	23
100.00 % Callback - System.Windows.Forms.NativeWindow.Callback(IntPtr, Int32, IntPtr, IntPtr)	0	0	23
56.52 % System.Windows.Forms.Timer.TimerNativeWindow.WndProc...	0	0	13
43.48 % WndProc - System.Windows.Forms.Control.ControlNativeWindow.WndProc(M	0	0	10

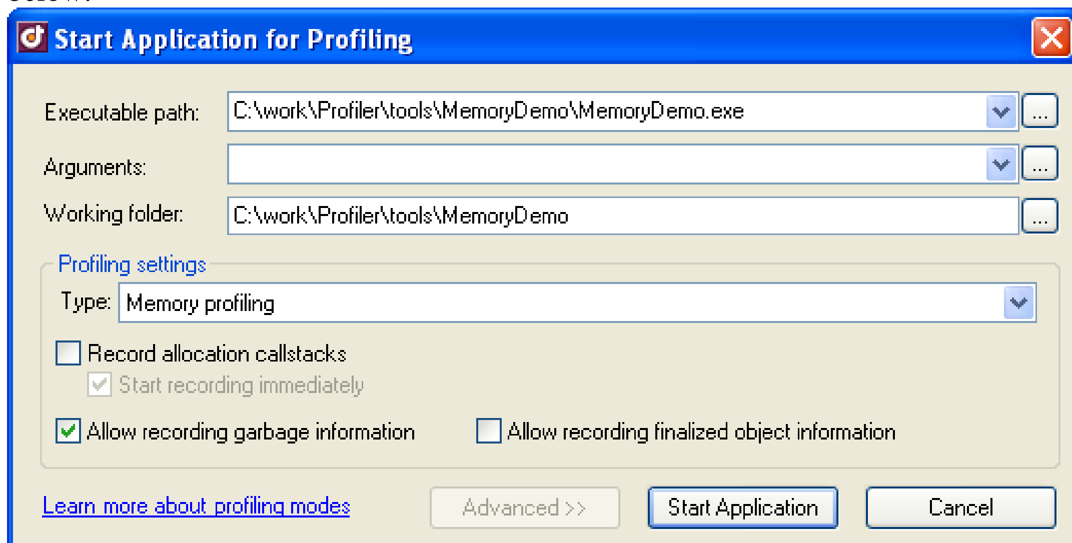
10. So, what is the bottom line here? You know dotTrace can be trusted to examine the current memory state of your applications from multiple points of view.

Memory Profiling: Memory Difference Mode

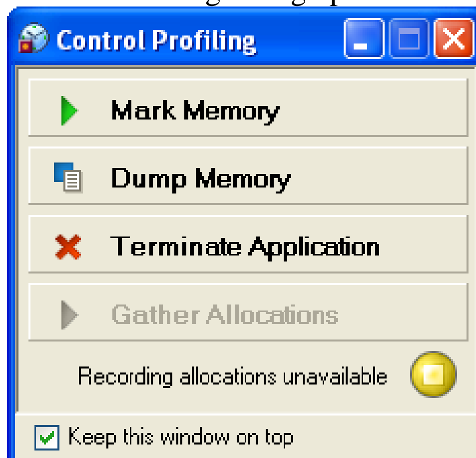
About this profiling mode
To view the difference between two application memory states, you can mark the start and the end of a time interval, and then capture a snapshot containing the difference. A *difference snapshot* lets you see how much new memory was allocated and how much was released during the marked time interval.

Step 1. Launch MemoryDemo application from dotTrace

1. Download the MemoryDemo application package from http://www.jetbrains.com/profiler/documentation/memorydemo.zip. Unzip the file to your computer.
2. Start dotTrace.
3. On the Welcome screen, click the "Profile application" link. In the following dialog, enter the MemoryDemo application execution path and select the options as shown below:

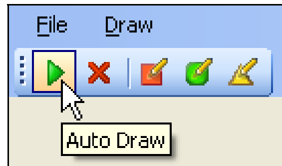


4. Click the **Start Application** button. The MemoryDemo application starts, and the Control Profiling dialog opens:

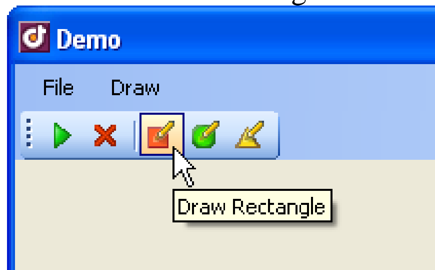


Step 2. Profile it

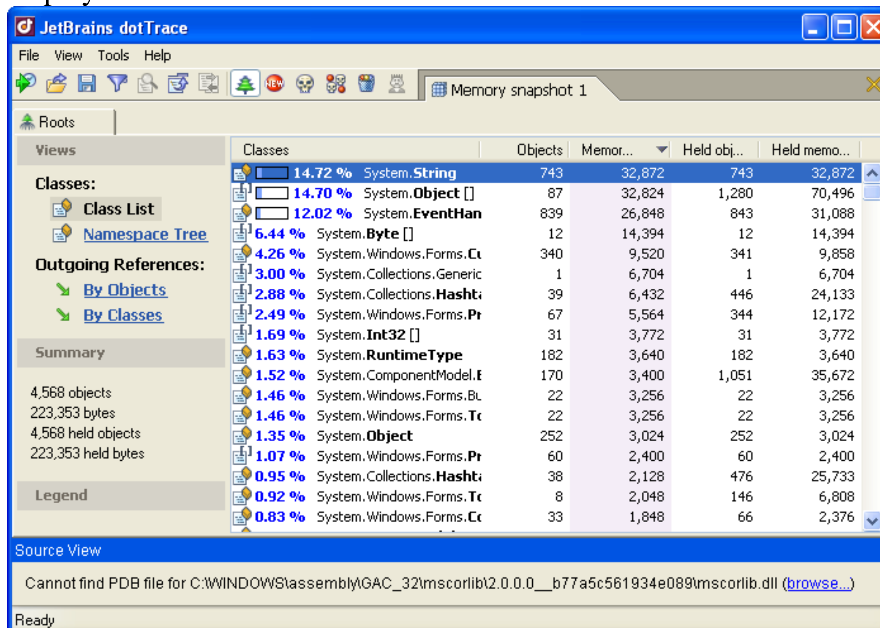
1. This demo application lets you draw simple shapes such as triangles, rectangles, and ellipses. We will draw a few of each type, erase them, and then see what objects remain in memory that *shouldn't be*.
2. Click the Auto Draw button:

   The application will draw a few random shapes.
3. After a few items of each type are drawn, click the button again to stop the drawing.
4. In performing *memory difference profiling*, we are interested in seeing the difference between a Point A and a Point B of our application. We will do this with the manual shape drawing mode.
   So, to mark "point A," click the **Mark Memory** button in the Control Profiling dialog.
5. Click the Draw Rectangle button in the MemoryDemo application:

   Enter a number, for example, 17. The application will draw 17 random rectangles.
6. Mark "Point B" by clicking the **Get Snapshot** button in the Control Profiling dialog. The memory snapshot representing the difference between Point A and Point B displays:

   NOTE: Do NOT close the demo application just yet.

Step 3. See what Memory Difference is about

1. Snapshots created in this mode will have the Memory Difference toolbar enabled:

   These buttons represents filters that let you see the following types of objects:
   - **Live** objects – all objects currently allocated
   - **New** objects – all live objects created after Point A
   - **Dead** objects – all objects deleted before Point B
   - **New and Dead objects difference**
   - **Garbage Objects** – objects created after Point A and deleted before Point B
   - **Finalized Objects** – objects deleted by the finalizer (unavailable in this scenario)

   We will use them later to check the states of some objects we created.
2. Switch to the Namespace Tree view:

3. Select the MemoryDemo namespace node and click *Ctrl+T* to open it in a new tab:

Classes	Objects
26.17 % MemoryDemo.DemoControl.RectangleDraw	24
17.76 % MemoryDemo.DemoForm	1
13.24 % MemoryDemo.DemoCountForm	1
12.62 % MemoryDemo.DemoControl.TriangleDraw	9
11.21 % MemoryDemo.DemoControl.IDraw []	2
9.81 % MemoryDemo.DemoControl.EllipseDraw	9
9.19 % MemoryDemo.DemoControl	1

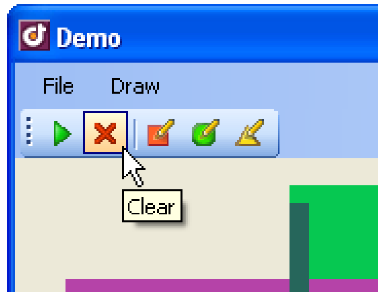
4. The numbers of objects of each shape (rectangle, triangle, and ellipse), as reported here, correspond to the ones provided by the application (see the running application to check these). However, the objects we created manually, after Point A, should show up as New objects.
5. Click the **New** button on the Memory Difference toolbar:

Classes	Objects
43.75 % MemoryDemo.DemoControl.RectangleDraw	17
31.25 % MemoryDemo.DemoCountForm	1
25.00 % MemoryDemo.DemoControl.IDraw []	1

   As you can see, exactly 17 RectangleDraw objects were created after we marked memory.

Step 4. Perform more actions in demo application

1. To see the difference between New and Dead objects, and some Garbage Collection information, we need to erase and draw some more shapes.
2. Switch back to the MemoryDemo application and mark memory again (click the **Mark Memory** button in the Control Profiling dialog).
3. Click the **Clear** button:

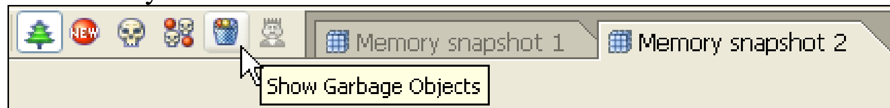


All objects are erased.

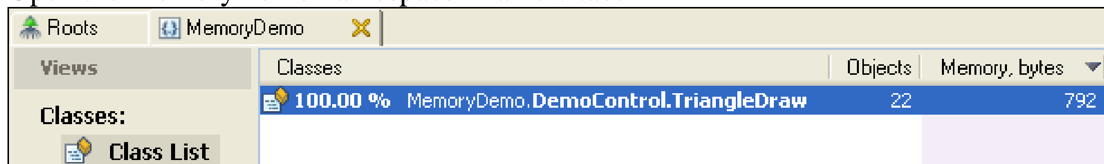
4. Draw several triangles (use the **Draw Triangle** button). We use 22 in this example.
5. Now, erase these triangles by clicking the **Clear** button again.
6. To capture the second memory snapshot, click the **Get Snapshot** button in the Control Profiling dialog. The second snapshot is opened in addition to the first one:

Step 5. Analyze second memory snapshot

1. Let's see the recorded Garbage objects. Click the **Show Garbage Objects** button on the Memory Difference toolbar:

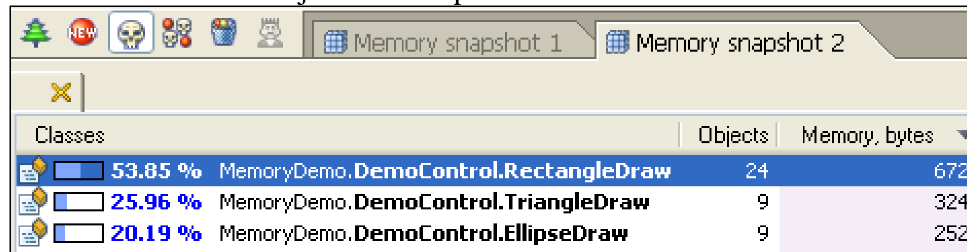


2. For easier navigation to the classes we want to see, switch to the Namespace Tree view.
3. Open the MemoryDemo namespace in a new tab:

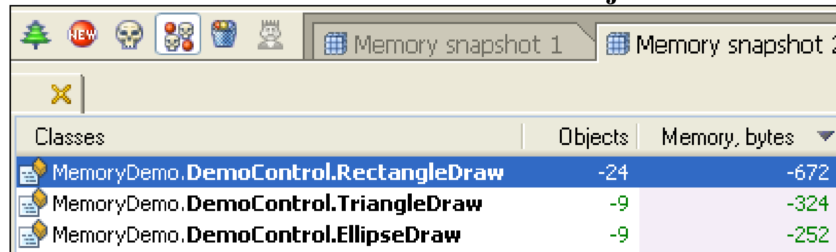


4. This shows that exactly 22 TriangleDraw objects were collected by the garbage collector between Point A and Point B.
5. But where are the other 23 Rectangles, 9 Ellipses, and 9 Triangles we erased also? Well, they *were* deleted after we marked memory; however, they were created *before* we marked memory. This is why they can be seen as **Dead** objects.
6. Click the **Show Dead Objects** button on the Memory difference toolbar:

The 24 + 9 + 9 dead objects show up here:



7. We can also see them in the **New and Dead Objects Difference** filter:

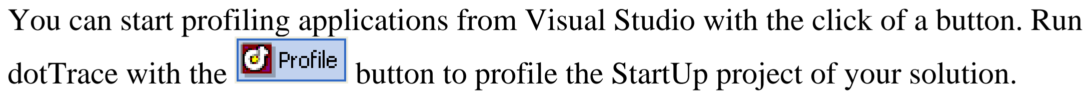


The figures are negative since the number of dead objects is subtracted from the number of new ones for each class.

8. So, what is the moral of this story? If your application seems to fail to delete some objects, possibly leading to memory leaks, profiling for Memory Difference may be the right answer.

Other features

Integration with Microsoft Visual Studio

You can start profiling applications from Visual Studio with the click of a button. Run dotTrace with the  button to profile the StartUp project of your solution.

Additional performance profiling modes

dotTrace offers two additional performance profiling modes: *Sampling profiling* (as opposed to *Tracing profiling*) and Routine thread time measurement (as opposed to *Wall time* measurement). A total of 4 possible profiling modes are available by combining

- **Sampling profiling** is a profiling method which is up to 30 times faster than regular tracing profiling, but at the expense of lower accuracy. It is extremely useful for at least two profiling scenarios: first, quickly getting a general idea of your application's performance, and second, profiling for extensive periods of time, up to many hours long.
- **Routine thread time** is measured by a thread-specific timer which is paused when its thread is paused. Using this mode, dotTrace can measure the contribution of individual threads in multithreaded applications, reducing interference between threads.

ASP.NET Applications Profiling

dotTrace also profiles ASP.NET applications running on IIS, versions 5.x and 6.0, or on the ASP.NET Development Server. Simply click **Profile Web Application** on the Welcome screen, specify the start page URL of your web application, and profile it the same way as any desktop program.

Command Line Options

If you prefer using a command line to control dotTrace, you can profile applications, open snapshots and generate reports from any batch script by using the command line options provided with dotTrace (the detailed list of options is provided in the online help).

Profiling API

dotTrace allows you to control its profiling functions from within the application being profiled, with the help of its Profiling API.

To use the API for profiling your applications:

1. Reference **JetBrains.dotTrace.Api.dll** (located in the dotTrace installation directory) in your application project
2. Surround the source code you want to profile with the following pattern:

```
JetBrains.dotTrace.Api.CPUProfiler.Start();
//Here goes the code which you want to profile
JetBrains.dotTrace.Api.CPUProfiler.StopAndSaveSnapShot();
```

Start the profiled application from within dotTrace, and in the Control Profiling dialog, clear the **Start profiling immediately** checkbox.

Additional Information

You can find the general product overview and the detailed list of features at the official product site at http://www.jetbrains.com/profiler.

Contacts

Ann Oreshnikova	**Alexander Morozov**
Marketing Director JetBrains, s.r.o. E-mail: pti@jetbrains.com	Marketing Manager JetBrains, s.r.o. E-mail: alexander.morozov@jetbrains.com

Thank you for reviewing and considering JetBrains dotTrace!