dotfuscator

Professional Edition &
Dotfuscator for Marketplace Apps

User's Guide

Version 4

PreEmptive Solutions

1 Table of Contents

2.1 Introduction

Why Obfuscate?

Programs written for .NET are easy to reverse engineer. .NET applications compile to a high-level, expressive file syntax called MSIL (Microsoft Intermediate Language) that contains method and variable names and can be easily decompiled back into source form.

Attackers can use freely available decompilers to easily see the source of any .NET application, exposing software licensing code, copy protection mechanisms, and proprietary business logic - whether it's legal or not. Anyone can peruse the details of a software application for security flaws to exploit, unique ideas to steal, features to crack, or worse.

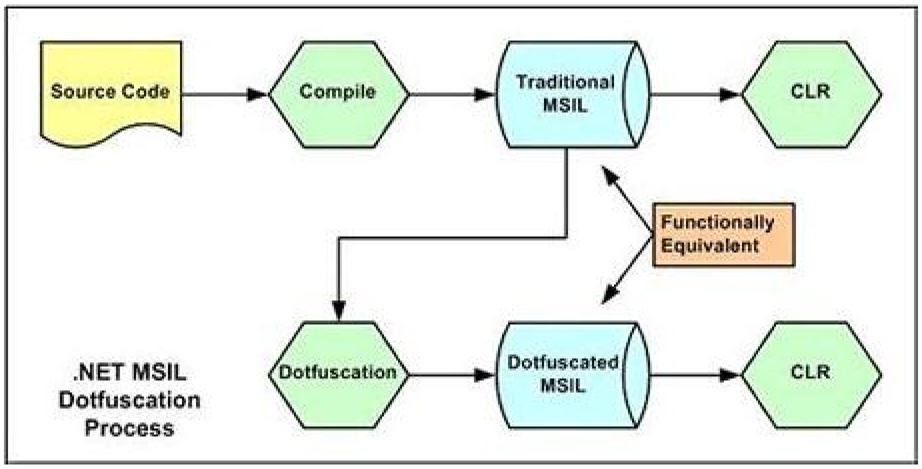
This is not the only option, though. Obfuscation is a technique that provides seamless renaming of symbols in assemblies as well as other tricks to foil decompilers. Properly applied obfuscation increases protection against decompilation by orders of magnitude, while leaving the application intact.

When an obfuscator tool goes to work on readable program instructions, a side effect is the output will confuse a human interpreter and break the decompiler the human interpreter was using. While the executable logic is preserved, the reverse semantics are rendered non-deterministic. As a result, attempts to reverse-engineer the instructions fail because the translation is ambiguous. Deep obfuscation creates a myriad of decompilation possibilities, some of which produce incorrect logic if recompiled. The decompiler, as a computing machine, has no way of knowing which of the possibilities could be recompiled with valid semantics. Humans write and employ decompilers to automate decompilation algorithms that are too challenging for the mind to follow. It is safe to say that any obfuscator that confuses a decompiler poses even more deterrence to a human attempting the same undertaking.

Post-build Obfuscation

It is important to understand that Dotfuscator is a "post-build" tool - it works by modifying already-compiled application assemblies. The development environment and tools do not change to accommodate obfuscation, and source code is never altered, or even read, in any way. Obfuscated binaries are functionally equivalent to traditional binaries and will execute on the Common Language Runtime (CLR) with identical results. (The reverse, however, is not true. Even if it were possible to decompile strongly obfuscated MSIL, it would have significant semantic disparities when compared to the original source code.)

The following illustration shows the flow of the Dotfuscation process:



That means you follow all your normal processes to develop and build your application, and then once you have compiled binaries you configure Dotfuscator to work with them, and run Dotfuscator's build process against them. Dotfuscator will take the binaries as input, perform the obfuscation per your configuration settings, and generate *the same set of binaries* as output - but those binaries will be obfuscated.

Most often, the process of developing and obfuscating an application looks like this:

1. Follow normal development and QA procedures for most of the project lifecycle
2. Somewhere past the half-way point but well before release, use the Dotfuscator UI (in Visual Studio or on its own) to configure Dotfuscator to work with your application, using basic smoke testing to ensure that the obfuscation hasn't caused obvious issues with the application
3. Integrate Dotfuscator into your build process, using our Visual Studio integration, MSBuild task, or straight command-line calls
4. Use the obfuscated build in QA for the remainder of the project

2.1.1 Dotfuscator Editions

Professional

Dotfuscator Professional Edition is the most capable, most powerful version of Dotfuscator. It is designed for organizations that produce commercial and enterprise applications. Dotfuscator Professional Edition provides superior protection to foil decompilation, advanced size reduction to conserve memory and improve load times, deep Visual Studio integration for seamless configuration, incremental obfuscation to release patches, watermarking to uniquely tag assemblies, and phone and technical support.

Community Edition

Dotfuscator Community Edition is a free version included with Visual Studio that offers basic obfuscation. Its main purpose is to rename identifiers, discouraging reverse engineering. Dotfuscator Community Edition incorporates advanced technologies to facilitate this protection and achieve some size reduction due to renaming to trivial identifiers.

Dotfuscator Community Edition does **not**:

- Dotfuscate managed code meant to run inside Microsoft SQL Server.
- Integrate deeply or operate separately from Visual Studio.
- Support Managed C++ applications.
- Support ClickOnce, APPX, or XAP packaging

See http://www.preemptive.com/products/dotfuscator/compare-editions for additional differences.

If you need to go beyond these limitations, contact PreEmptive Solutions for more information about Dotfuscator Professional Edition.

This guide does not cover Dotfuscator Community Edition. Please see the guide included with that version for additional information.

Dotfuscator for Marketplace Apps

Dotfuscator for Marketplace Apps includes most of the features available in Dotfuscator Professional Edition, but only for applications targeting Windows Phone 7 or 8, and/or Windows RT via the Windows Store. and is intended to supercede Dotfuscator for Windows Phone Edition. If support for other application types is required, contact our sales department to inquire about Dotfuscator Professional Edition.

If a feature behaves differently in Dotfuscator for Marketplace Apps than in Dotfuscator Professional Edition, it will be flagged with a DMA icon.

2.2 Getting Started

This section shows you how to get started using Dotfuscator for obfuscation. It starts with registering the software, then reviews the three standard ways to use Dotfuscator (GUI, command line, Visual Studio), then works through examining the obfuscated assemblies.

2.2.1 Registering and Activating Dotfuscator

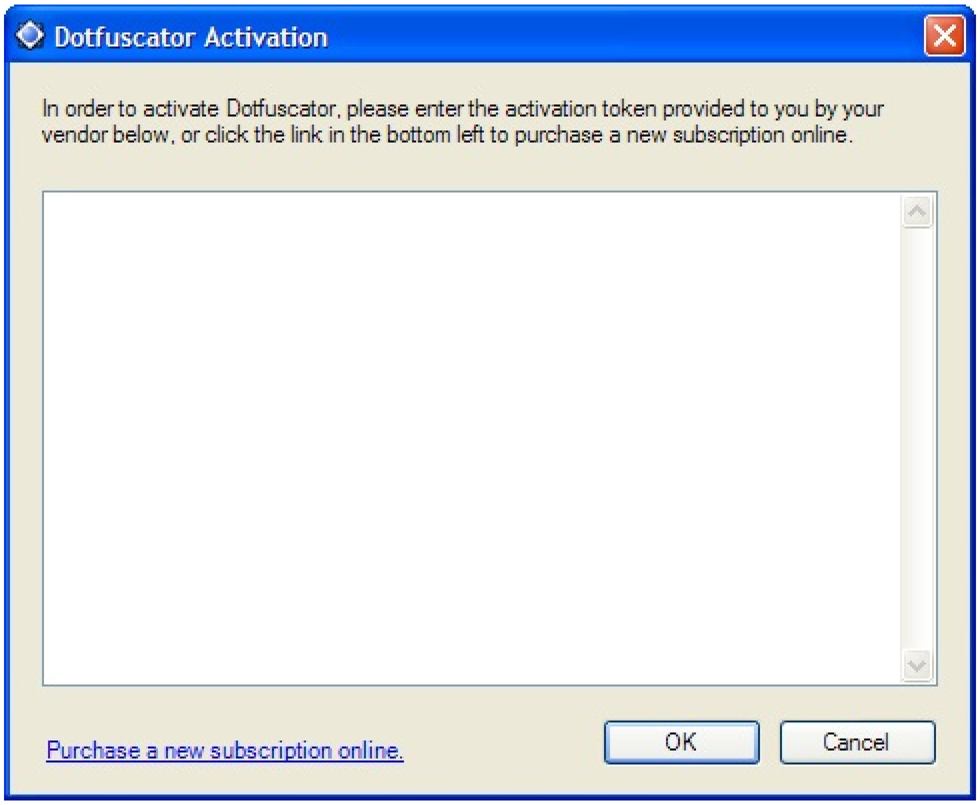
The first time you use Dotfuscator you will be prompted to register, which is a two-step process. In the first step, you fill out the registration form; Dotfuscator will then send this information to the registration server via or via the web, depending on your choices. Upon successful receipt, the server will generate a serial number and confirmation code. This information will be sent back to you, using the address that you provided.

After this step is completed, the next time you run Dotfuscator, you will be prompted for your serial number and confirmation code. Once you enter this information and it is validated, registration is complete.

If you purchased a subscription license (as opposed to a perpetual license) of Dotfuscator, then after registration Dotfuscator will attempt to download your registration and subscription information from PreEmptive's servers. If your subscription has expired, you will be prompted to renew your subscription. If the servers cannot be contacted you will be prompted to provide your activation token, which was supplied to you by your vendor.

Graphical Offline Activation

When using Dotfuscator's graphical user interface, if PreEmptive's subscription servers cannot be contacted and you have not activated your subscription or your subscription has expired, you will be prompted to provide your activation token in the offline activation dialog.


Dotfuscator Activation

In order to activate Dotfuscator, please enter the activation token provided to you by your vendor below, or click the link in the bottom left to purchase a new subscription online.

If you select **Cancel** from the Activation Dialog Dotfuscator will load in reduced functionality mode. In this mode, you will only be able to access the Activation Dialog. You can do this by selecting **Help > Activate Dotfuscator**.

Dotfuscator will also load in reduced functionality mode if your subscription has expired and you have not renewed it. Dotfuscator's status bar will alert you to the status of your subscription. Additionally, the Activation Dialog and Dotfuscator's status bar contain links to purchase or renew your subscription.

Upon purchasing or renewing your subscription, you may simply re-launch Dotfuscator and it will attempt to download your updated subscription information from PreEmptive's servers, and Dotfuscator will once again operate in full functionality mode. You will only need to use the Activation Dialog if PreEmptive's servers cannot be contacted.

Command Line Offline Activation

When using Dotfuscator's command line interface, if PreEmptive's subscription servers cannot be contacted and you have not activated your subscription or your subscription has expired, Dotfuscator will exit with a message informing you how to provide your activation token via the `/offlineactivation` command line option.

Upon purchasing or renewing your subscription, you may simply re-launch Dotfuscator and it will attempt to download your updated subscription information from PreEmptive's servers. You will only need to use the **`/offlineactivation`** command line option if PreEmptive's servers cannot be contacted.

2.2.2 Standalone GUI Quick Start

This section shows you how to use Dotfuscator's standalone GUI. For a complete guide to Dotfuscator's User Interface, see Graphical User Interface Reference.

Step 1 – Create a Default Project

- Launch Dotfuscator from the Windows Start menu.
- Select **Create New Project** and click **OK**. The Dotfuscator main project window displays with the *Input* tab selected.
- Select the assembly you would like to obfuscate:
  - Click **Browse**.
  - Browse to:

Path
`C:\Program Files (x86)\PreEmptive Solutions\Dotfuscator Professional Edition 4.x\samples\cs\GettingStarted\bin\Debug`

  - and select **GettingStarted.exe**.
  - Click **Open**.
- The path to the executable populates the list box under *Input Files*.
- Select **File > Save Project** to save the project.
- In the *Save Project* dialog, navigate to:

Path
`C:\Program Files (x86)\PreEmptive Solutions\Dotfuscator Professional Edition 4.x\samples\cs\GettingStarted\`

- In the *File Name* field, enter **Dotfuscator.xml** and click **Save**.

Step 2 – Build the Project

- Click on the **Settings** tab and select *Build Settings*. The *Destination Directory* field is populated by default as: *${configdir}\Dotfuscated*.

Note: **${configdir}** is a variable that holds the path to your Dotfuscator configuration file.

- The project is ready to be obfuscated. Click the **Build** button on the toolbar. Wait a few moments as Dotfuscator builds an obfuscated version of the *HelloWorld* application. The obfuscated assembly is now stored in:

Path

```
C:\Program Files (x86)\PreEmptive Solutions\Dotfuscator Professional Edition
4.x\cs\GettingStarted\Dotfuscated
```

You can now go to the *Output* tab and browse the obfuscated symbols, or look at the renaming map file (*map.xml*) that Dotfuscator created in the output directory. Or, run the obfuscated application if you wish.

Next, with a little more configuration, we can use some of Dotfuscator's more powerful features.

Step 3 – Configure the Project

- Click the ***Settings*** tab and select ***Global Options***.
  - In the ***General*** section, set the *Build Progress* property to *Verbose*. This causes Dotfuscator to provide additional information about its progress during execution at the bottom of the *build* tab.
  - In the ***Advanced*** section, Set the *Emit Debugging Symbols* to *JIT Optimization; Sequence Points from PDB*. Setting this property tells Dotfuscator to create a symbol file in PDB format for each output assembly. Debuggers can use these files to provide useful information in a debugging session. Typically, they contain information such as line numbers, source file names, and local variable names. The PDB files are placed in the *output* directory with the output assemblies.
  - In the ***Feature*** section, Set the values for *Disable Renaming*, *Disable Control Flow*, *Disable String Encryption*, and *Disable Removal* to *No*. You have fine grained control over which transforms Dotfuscator will apply to your assemblies; these are the features we will configure and use in the next steps.
- Still on the ***Settings*** tab, select ***Reports > Renaming***. Check the *Output as HTML* checkbox to get a useful report containing renaming information and statistics on your application. This report will output into the same directory as the map file. The default location is *${configdir}\Dotfuscated\Map.html.*
- Click the ***Rename*** tab, then ***Options*** sub tab. Check *Use Enhanced Overload Induction*. This feature allows up to 15% more redundancy in method and field renames. Since overloading on method return type or field type is typically not allowed in source languages (including C# and VB), this further hinders decompilers.
- Click the ***String Encryption*** tab. String encryption is inclusion based, therefore you must mark the assembly's checkbox at the root of the tree shown in the left pane to include all methods in the input assembly. String Encryption scrambles the strings in your application. For example, someone looking to bypass your registration and verification process can search for the string where your program asks the user for a serial number. When they find the string, they locate instructions near it and alter the logic. String Encryption makes this difficult to do, because their search will come up empty.

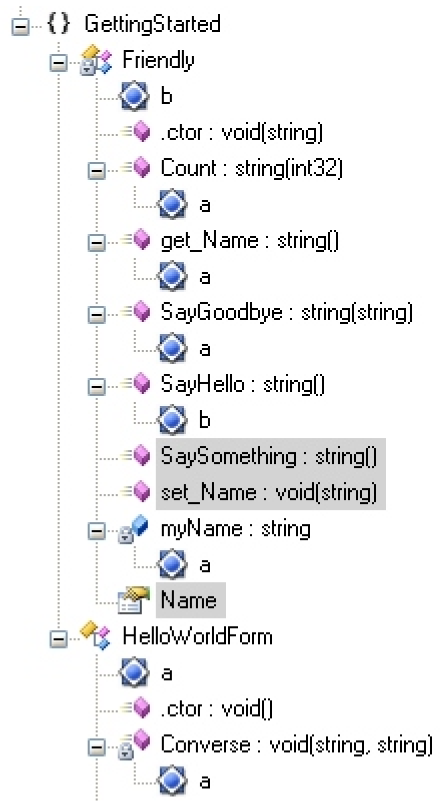
Step 4 – Rebuild the Project

- Click **Build**; the project is ready to be re-obfuscated. As before, the obfuscated assembly is stored in:

Path
`C:\Program Files (x86)\PreEmptive Solutions\Dotfuscator Professional Edition 4.x\samples\cs\GettingStarted\dotfuscated`

Step 5 – Browse the Output

- Click the *Output* tab. Now, you can navigate a tree that shows how Dotfuscator obfuscated your code.
- Expand the root tree and all sub-trees. Notice the **blue** diamond shaped icons. These are the renamed methods and fields. The parents of each of these icons display their original names. Dotfuscator has renamed each method and field to make it almost impossible to decipher the purpose of each method. This severely impacts the process of reverse engineering the code.



```
{} GettingStarted
    Friendly
        b
        .ctor : void(string)
        Count : string(int32)
            a
        get_Name : string()
            a
        SayGoodbye : string(string)
            a
        SayHello : string()
            b
        SaySomething : string()
        set_Name : void(string)
        myName : string
            a
        Name
    HelloWorldForm
        a
        .ctor : void()
        Converse : void(string, string)
            a
```

- Notice the currently highlighted **`SaySomething`** and **`set_Name`** methods, as well as the **`Name`** property. Dotfuscator determined these items are not used in this application. As a result, Dotfuscator's Pruning feature is able to remove them, resulting in a more compact application.

Next Steps

Now that you have successfully obfuscated using the GUI, you can see how to use the command line interface to do the same things. Or you can examine the obfuscated output assembly in detail and see how effective the obfuscation was.

2.2.3 Command Line Quick Start

This section demonstrates how to use the command line interface to obfuscate using the same settings as in the Standalone GUI Quick Start. For a complete guide to Dotfuscator's command line, see Command Line Interface Reference.

You can start Dotfuscator from the command line using the following syntax:

Command Line Quick Start

```
dotfuscator [options] [configfile]
```

The command line options are documented in the Command Line Options Summary. The configuration file is an XML document that specifies various options for Dotfuscator. When you ran the standalone GUI and filled in the various dialogs, you were populating a configuration file. All elements of the configuration file are documented in the Configuration File Reference.

Using Existing Configurations

You can feed previously created configuration files into the command line tool. For example, using the configuration file that you created in the last section, you can obfuscate from the command line using this command:

Using Existing Configurations

```
dotfuscator Dotfuscator.xml
```

Using Command Line Switches Only

Alternatively, you can Dotfuscate on the command line without a configuration file because most of the configuration options are available as command line switches. To get powerful obfuscation for our example assembly, all you need to do is specify your input assembly.

Using Command Line Switches Only

```
dotfuscator /in:GettingStarted.exe
```

- The **`in`** switch lets you specify a list of input assemblies separated by commas.
- By default, the output assembly is placed in a subdirectory of the working directory called **Dotfuscated**. You can override this with the **`out`** command line switch.
- By default, renaming is enabled and the renaming map file is called **map.xml**. It is also placed in the **Dotfuscated** subdirectory. You can override this with the **`mapout`** switch.
- By default, string encryption, control flow, and removal are turned on.

Using Advanced Command Line Switches

If you want to run the obfuscator from the command line with the same options that you set in the standalone GUI in the previous section, you need a command like this:

Using Advanced Command Line Switches

```
dotfuscator /in:GettingStarted.exe /debug:on /v /enha:on
```

- The **`in`** option is as before.
- The **`v`** option runs Dotfuscator in verbose mode.
- The **`debug`** option tells Dotfuscator to generate the debugging symbols for the obfuscated output assemblies.
- The **`enha`** option turns on Enhanced Overload Induction.

2.2.4 Visual Studio Integrated UI Quick Start

This section shows you how to use Dotfuscator from within Visual Studio. You can then include obfuscation as part of a Visual Studio solution build. A Dotfuscator project can accept input files from one or more other Visual Studio Projects (such as C# or VB.NET projects), or you can specify assemblies directly from a file browse dialog.

For a detailed guide to Dotfuscator's Visual Studio integration, see Using the Visual Studio Integrated User Interface under the Graphical User Interface Reference.

Additionally, an online demonstration is available at www.preemptive.com/demos.html.

DMA The Dotfuscator interface within Visual Studio is unavailable in Dotfuscator for Marketplace Apps.

Step 1 – Open the GettingStarted Solution with Visual Studio

Within Visual Studio, click on **Open Solution**, and browse to:

Getting Started

```
C:\Program Files (x86)\PreEmptive Solutions\Dotfuscator Professional Edition
4.x\samples\cs\GettingStarted\GettingStarted.sln
```

- The solution and project files are in Visual Studio 7.0 format. If you are using a later version of Visual Studio, you will be asked if it is OK for Visual Studio to upgrade the files. You should agree to the upgrade before continuing.
- The **GettingStarted** project displays in Solution Explorer. This is a C# project that you can use to build the **`GettingStarted`** executable.

Step 2 – Create a Dotfuscator Project

- Click on **Add New Project** from Visual Studio's **File** menu.
- In the *Add New Project* Dialog, click on **Dotfuscator Projects**. Click on the **Dotfuscator Project** icon and name the project *GetDotfuscated*. Click **OK** to create the project.
- A new Dotfuscator project called *GetDotfuscated* displays in the Solution Explorer. Use the Solution Explorer as the starting point for configuring Dotfuscator projects.
- To specify the input file, tell Dotfuscator to use the output from the *GettingStarted* project. Right click on the top level **GetDotfuscated** node and select **Add Project Output** from the context menu. This brings up the *Add Project Output* dialog.
- On the *Add Project Output* dialog, select the **GettingStarted** project from the project drop down, then select **Primary Output** from the output groups list. Click **OK**.
- Under the Dotfuscator project's **Input Assemblies** node within Solution Explorer, you should now see *GettingStarted.exe from GettingStarted (Active)*.

Step 3 – Build the Solution

- Right click on the **GetDotfuscated** project node in Solution Explorer and select **Properties** from the context menu. This brings up the property pages for your Dotfuscator project. Click on the **Configuration Properties > Build Settings** properties in the left navigation tree. The Output Directory is populated by default as: **`${configdir}\Dotfuscated`**.

Note: `${configdir}` is a variable that holds the path to your Dotfuscator configuration file.

- If you have installed the samples to the default location, the obfuscated assembly will be written to:

Building the Solution

```
C:\Program Files (x86)\PreEmptive Solutions\Dotfuscator Professional Edition
4.x\samples\cs\GettingStarted\GetDotfuscated\Debug\Dotfuscated
```

- The solution is ready to be built. Select **Build Solution** from the **Build** menu. The C# project builds first, then the Dotfuscator project. Dotfuscator's output displays in Visual Studio's output window.

Within Solution Explorer, you can bring up Dotfuscator's *Output Browser* by double clicking the **Output** node in the **GetDotfuscated** project. Here you can see the original and new names that Dotfuscator applied.

Next, with a little more configuration, we can use some of Dotfuscator's more powerful features.

Step 4 – Configure the Project Properties

- Right click on the **GetDotfuscated** project node in Solution Explorer and select **Properties** from the context menu.
- Select **Configuration Properties > Global Options** from the left tree. This brings up the property sheet for your Dotfuscator project's global options.
    - In the **Feature** section, set the values for *Disable Renaming*, *Disable Control Flow*, *Disable String Encryption*, and *Disable Removal* to *No*. You have fine grained control over which transforms Dotfuscator will apply to your assemblies; these are the features we will configure and use in the next steps.
    - In the ***General*** section, Set the *Build Progress* property to *Verbose*. This causes Dotfuscator to provide additional information in the Visual Studio output window during builds.
    - In the **Advanced** section, set the *Emit Debug Symbols* property to *JIT Optimization; Sequence Points from PDB* (this is the default for Debug project configurations). Setting this option tells Dotfuscator to create a symbol file in PDB format for each output assembly. Debuggers use these files to provide useful information in a debugging session. Typically, they contain information such as line numbers, source file names, and local variable names. The PDB files are placed in the output directory with the output assemblies.
- Select **Configuration Properties > Reports > Renaming**. Check the *Output as HTML* checkbox to get a useful report containing renaming information and statistics on your application. This report will output into the same directory as the map file. The default location is **`${configdir}\Dotfuscated\Map.html`**.
- Select **Configuration Properties > Reports > Removal**. In the Removal Report File editor, specify `${configdir}\Dotfuscated\removal.xml` for the Removal Report File. Also, check *Output as HTML* to get a formatted report containing the removal information and statistics on your applications. This report outputs into the same directory as the **`removal.xml`** file.

Step 5 – Configure Obfuscation Settings

- In Solution Explorer, expand the **Configuration Options** folder in the Dotfuscator Project. Here you will see a node for each configurable obfuscation setting. A node is grayed out if the feature is disabled. To enable or disable a feature, you can either right click on its node and check or uncheck the disabled menu item, or you can set it using the Global Properties sheet as described in step 4. In this section, we are going to use renaming, control flow obfuscation, string encryption, and removal.
- Double click the **Renaming** node and click the **Options** sub tab. Check *Use Enhanced Overload Induction*. This feature allows up to 15% more redundancy in method and field renames. Since overloading on method return type or field type is typically not allowed in source languages (including C# and VB), this further hinders decompilers.
- Check **Output as HTML** to get a useful report containing renaming information and statistics on your application. This report outputs into the same directory as the map file. The default location is **`${configdir}\Dotfuscated\Map.html`**.
- Double click the **String Encryption** node. String encryption is inclusion-based, therefore, you must mark the assembly's checkbox at the root of the tree shown in the left pane to include all methods in the input assembly. String Encryption scrambles the strings in your application. For example, someone attempting to bypass your registration and verification process can search for the string where your program asks the user for a serial number. When they find the string, they can locate instructions near it and alter the logic. String Encryption makes this difficult to do, because their search will come up empty.

Step 6 – Rebuild the Solution

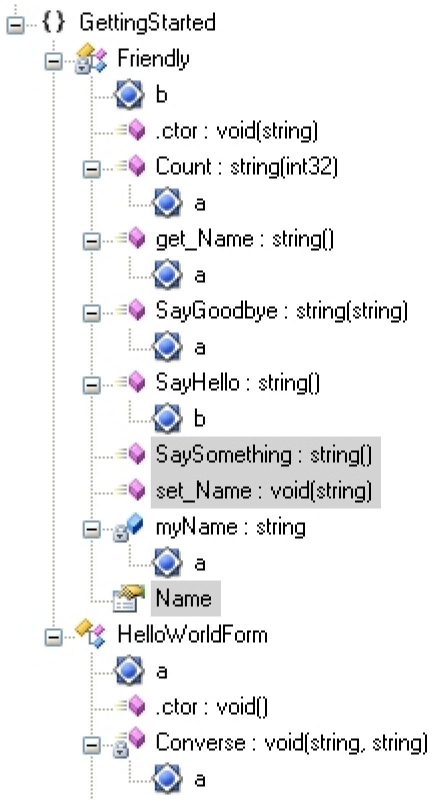
- Build the solution again. As before, the obfuscated assembly is stored in:

Rebuild the Solution

```
C:\Program Files (x86)\PreEmptive Solutions\Dotfuscator Professional Edition
4.x\samples\cs\GettingStarted\GetDotfuscated\Debug\Dotfuscated
```

Step 7 – Browse the Output

- Double click the **Output** node in the Dotfuscator project. You can now navigate a tree that shows how Dotfuscator obfuscated your code.
- Expand the root tree and all sub-trees. Notice the **blue** diamond shaped icons. These are the renamed methods and fields. The parents of each of these icons display their original names. Dotfuscator renamed each method and field to make it almost impossible to decipher the purpose of each method. This severely impacts the process of reverse engineering the code.


```
GettingStarted
  Friendly
    b
    .ctor : void(string)
    Count : string(int32)
      a
    get_Name : string()
      a
    SayGoodbye : string(string)
      a
    SayHello : string()
      b
    SaySomething : string()
    set_Name : void(string)
    myName : string
      a
    Name
  HelloWorldForm
    a
    .ctor : void()
    Converse : void(string, string)
      a
```

- Notice the currently highlighted `SaySomething` and `set_Name` methods, as well as the `Name` property. Dotfuscator determined these items are not used in this application. As a result, Dotfuscator's Pruning feature removes them, resulting in a more compact application.

Step 8 – Browse the Reports

- Select the **GetDotfuscated** Project node in Solution Explorer and then click **View > Dotfuscator** in the menubar. There are three menu items here that are enabled whenever there are HTML versions of the map and removal reports available for viewing. Clicking on them displays the reports in your default browser.

Next Steps

Now that you have successfully obfuscated an application inside of Visual Studio, you can use the command line interface to do the same things. Or you can examine the obfuscated output assembly in detail and see how effective the obfuscation was.

To continue learning about Dotfuscator's Visual Studio integration, see Using the Visual Studio Integrated User Interface.

2.2.5 Observing and Understanding Obfuscated Output

Step 1 – Using a Disassembler

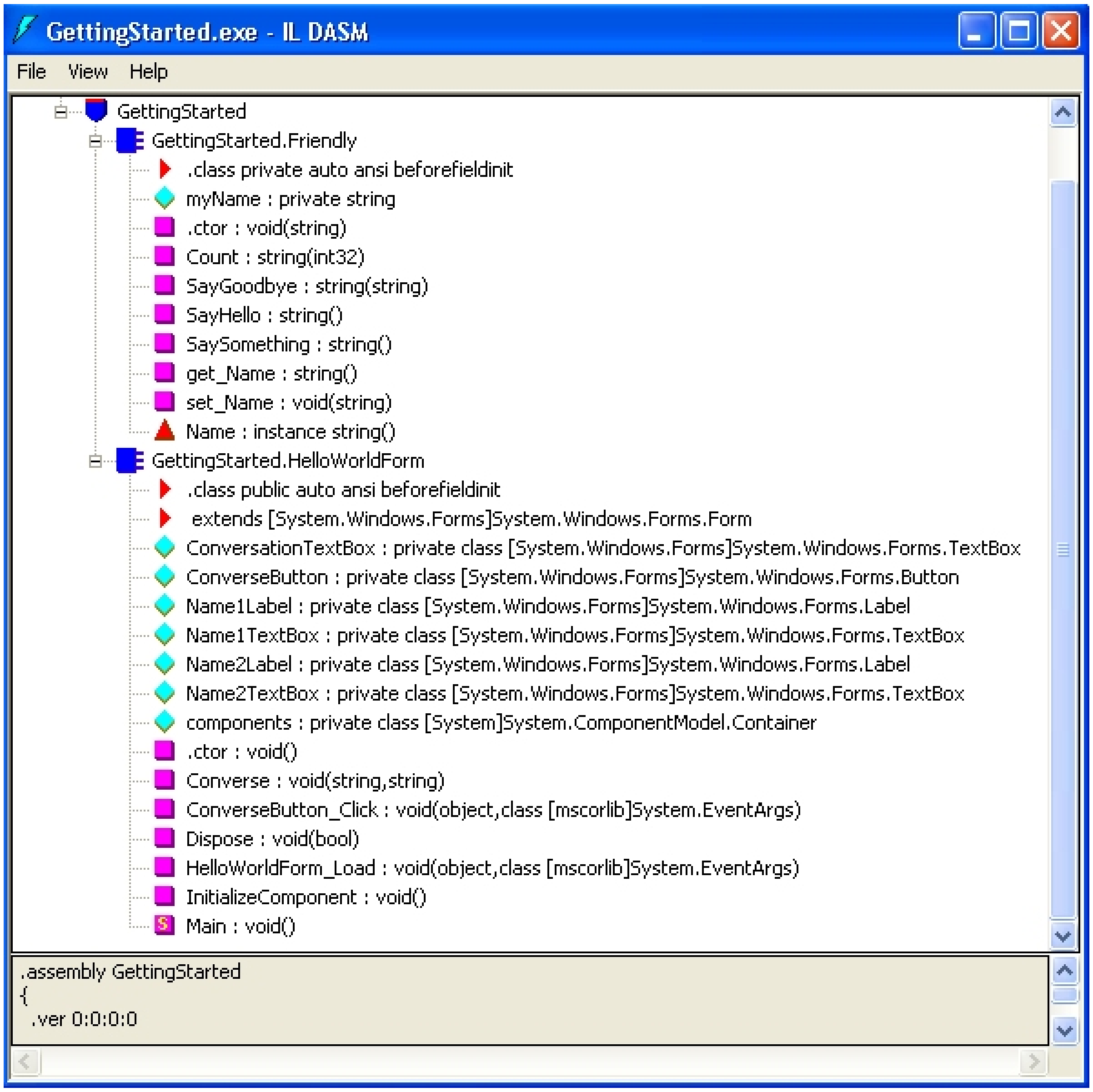
The .NET Framework SDK ships with a disassembler utility called ildasm that allows you to decompile .NET assemblies into IL Assembly Language statements. To start ildasm , make sure that the .NET Framework SDK is installed and in your path. Type **ildasm** on the command line.

Note: If this does not work and Visual Studio is installed, then ildasm is not in your path. To open a Visual Studio command prompt, click **start > Visual Studio [version] > Visual Studio Tools > Visual Studio [version] Command Prompt**. Type **ildasm**.

- Click **File > Open** and browse to:

Path
`C:\Program Files (x86)\PreEmptive Solutions\Dotfuscator Professional Edition 4.x\samples\GettingStarted\bin\Debug`

- and select **GettingStarted.exe**.
- Click **Open**. A view of the disassembled assembly displays:



GettingStarted.exe - IL DASM

File View Help

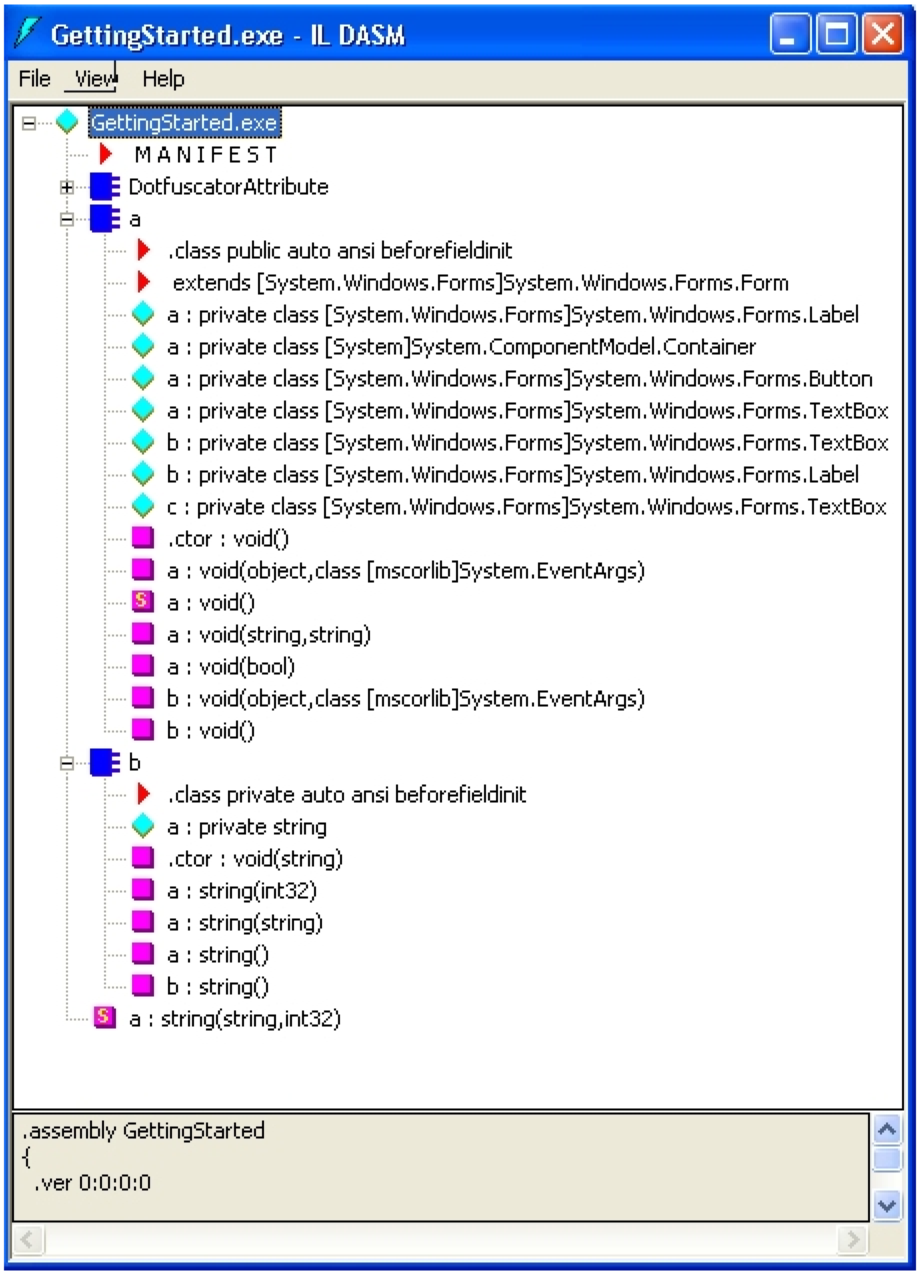
```
GettingStarted
  GettingStarted.Friendly
    .class private auto ansi beforefieldinit
    myName : private string
    .ctor : void(string)
    Count : string(int32)
    SayGoodbye : string(string)
    SayHello : string()
    SaySomething : string()
    get_Name : string()
    set_Name : void(string)
    Name : instance string()
  GettingStarted.HelloWorldForm
    .class public auto ansi beforefieldinit
     extends [System.Windows.Forms]System.Windows.Forms.Form
    ConversationTextBox : private class [System.Windows.Forms]System.Windows.Forms.TextBox
    ConverseButton : private class [System.Windows.Forms]System.Windows.Forms.Button
    Name1Label : private class [System.Windows.Forms]System.Windows.Forms.Label
    Name1TextBox : private class [System.Windows.Forms]System.Windows.Forms.TextBox
    Name2Label : private class [System.Windows.Forms]System.Windows.Forms.Label
    Name2TextBox : private class [System.Windows.Forms]System.Windows.Forms.TextBox
    components : private class [System]System.ComponentModel.Container
    .ctor : void()
    Converse : void(string,string)
    ConverseButton_Click : void(object,class [mscorlib]System.EventArgs)
    Dispose : void(bool)
    HelloWorldForm_Load : void(object,class [mscorlib]System.EventArgs)
    InitializeComponent : void()
    Main : void()
```

```
.assembly GettingStarted
{
 .ver 0:0:0:0
```

- To compare the currently shown, un-obfuscated *HelloWorld* application to the obfuscated version, start another copy of ildasm. This time browse to

Path
`C:\Program Files (x86)\PreEmptive Solutions\Dotfuscator Professional Edition 4.x\samples\GettingStarted\Dotfuscated.`

- and select **GettingStarted.exe**.
- Click **Open**.



```
GettingStarted.exe
    MANIFEST
    DotfuscatorAttribute
    a
        .class public auto ansi beforefieldinit
        extends [System.Windows.Forms]System.Windows.Forms.Form
        a : private class [System.Windows.Forms]System.Windows.Forms.Label
        a : private class [System]System.ComponentModel.Container
        a : private class [System.Windows.Forms]System.Windows.Forms.Button
        a : private class [System.Windows.Forms]System.Windows.Forms.TextBox
        b : private class [System.Windows.Forms]System.Windows.Forms.TextBox
        b : private class [System.Windows.Forms]System.Windows.Forms.Label
        c : private class [System.Windows.Forms]System.Windows.Forms.TextBox
        .ctor : void()
        a : void(object,class [mscorlib]System.EventArgs)
        a : void()
        a : void(string,string)
        a : void(bool)
        b : void(object,class [mscorlib]System.EventArgs)
        b : void()
    b
        .class private auto ansi beforefieldinit
        a : private string
        .ctor : void(string)
        a : string(int32)
        a : string(string)
        a : string()
        b : string()
    a : string(string,int32)
```

```
.assembly GettingStarted
{
  .ver 0:0:0:0
```

- Place each ildasm window side-by-side. Compare and contrast both figures.
- Notice the un-obfuscated disassembly contains names of methods that are fairly understandable. For example, it is safe to assume that the `ConverseButton_Click: void (object, class [mscorlib]System.EventArgs)` method is called when the **Converse** button is clicked. Now look at the obfuscated version. Which method is called when the converse button is clicked? It is hard to tell. Also notice the missing `SaySomething` method. It was removed because the method wasn't being used anywhere in the code.

- Double-click the methods **`SayHello:string()`** from the original assembly and **`a:string()`** from the obfuscated assembly. These two methods are the same; however, when examining the disassembled IL code further, notice that the strings have been encrypted in the obfuscated version to make the code difficult to read. For example, locate the following line in the un-obfuscated version:

Un-Encrypted String:

```
IL_0000:  ldstr      "Hello, my name is "
```

Now view the obfuscated version, and try to find the above string. If you're having trouble finding it, it's because it's encrypted and looks like the following:

Encrypted String:

```
IL_0000: ldstr bytearray (09 42 26 44 29 46 2B 48 26 4A 67 4C 6D 4E 22 50
                          28 52 73 54 3B 56 36 58 34 5A 3E 5C 7D 5E 36 60
                          12 62 43 64 )
```

You can imagine how confusing this can be for attackers who are trying to reverse-engineer the code, especially with more complex applications.

Step 2 –Decompiling

If you're thinking your source code is accessible only to a small circle of technical folks who actually know IL Assembly Language, think again. You can take this a step further and actually recreate the source code from our application by using a decompiler such as Reflector. These utilities can decompile a .NET assembly directly back to a high level language like C#, VB .NET, or Managed C++.

In this section we use Reflector for .NET, from http://www.red-gate.com/products/reflector/.

Running .NET Reflector against the Dotfuscated **GettingStarted.exe** file and trying to examine a method such as **`a()`** displays the following:

Run .NET Reflector Against GettingStarted.exe

```
This item appears to be obfuscated and can not be translated.
```

Thus, Dotfuscator successfully prevented a major decompiler from reverse engineering your Dotfuscated code.

2.3 Understanding Obfuscation with Dotfuscator

Dotfuscator provides a multi-part approach to application protection. Each of those parts provides an important piece of an overall protection strategy - if one part is compromised, the other parts continue to provide effective protection, making it harder and harder for an attacker to accomplish their goal. And not only are there multiple parts, but each part has advanced features that go beyond the simple techniques used by other obfuscation products.

The first part is Obfuscation, which sub-divides into:

- Renaming
- Control Flow
- String Encryption

The remaining parts strengthen overall application protection, beyond obfuscation:

- Pruning
- Tamper Detection and Defense
- Shelf Life
- Watermarking

Dotfuscator also provides features that help you build, deploy, and debug obfuscated assemblies. These include:

- Linking
- Incremental Obfuscation
- Debugging Obfuscated Code

Finally, there are a number of additional concepts that apply to projects that use individual language features or target specific assemblies. You can read more about these under Advanced Topics.

2.3.1 Protection Concepts

2.3.1.1 Renaming

Dotfuscator is capable of renaming all classes, methods, and fields to short (space-saving) names. In addition to making decompiled output much more difficult to understand, it also makes the resulting executable smaller in size.

Most commercial obfuscators employ a renaming technique that applies trivial identifiers that can be as short as a single character. As the obfuscator processes the code, it selects the next available trivial identifier for substitution. This seemingly simple renaming scheme has a key attribute: it cannot be reversed. While the program logic is preserved, the names become nonsense, hampering all attempts to understanding the code.

Dotfuscator uses a deeper form of obfuscation, developed for Dotfuscator and patented by PreEmptive Solutions, called Overload Induction™. Instead of substituting one new name for each old name, Overload Induction renames as many methods as possible to the same name. After this deep obfuscation, the logic, while not destroyed, is beyond comprehension. The following simple example illustrates the power of the Overload Induction technique:

Original Source Code Before Obfuscation

```
private void CalcPayroll(SpecialList employeeGroup) {
   while (employeeGroup.HasMore()) {
        employee = employeeGroup.GetNext(true);
        employee.UpdateSalary();
        DistributeCheck(employee);
    }
}
```

Reverse-Engineered Source Code After Overload Induction Dotfuscation

```
private void a(a b) {
    while (b.a()) {
        a = b.a(true);
        a.a();
        a(a);
    }
}
```

The example shows that the code is obfuscated and compacted, a positive side effect of renaming. For example, if a name is 20 characters long, renaming it to **`a()`** reduces its size by 95%. Renaming also saves space by conserving string heap entries. Renaming everything to "**`a`**" means that **`a`** is stored only once, and each method or field renamed to **`a`** can point to it. Overload Induction enhances this effect because the shortest identifiers are continually reused.

Many customers report a full third of all methods being renamed to "**`a()`**".

There are several distinct advantages to this methodology:

- Renaming makes decompiled output difficult to understand. Renaming to unprintable characters or illegal names in the target source language is futile since decompilers are equipped to re-rename such identifiers. Considering that Overload-Induction could make one of three method names "**`a()`**", understanding decompiled output is difficult.
- Overload-Induction has no limitations that do not exist in all renaming systems (such limitations are discussed later).
- Since overload-induction tends to use the same letter more often, it reaches into longer length names more slowly (e.g. **`aa`**, **`aaa`**, etc.). This also saves space.

Overload-Induction's patented algorithm determines all possible renaming collisions and only induces method overloading when it is safe to do so. The procedure is provably irreversible. In other words, it is impossible (even via running Overload-Induction again) to reconstruct the original method name relationships.

Dotfuscator also provides **Advanced Overload-Induction™** by allowing a method's return type or a field's type to be used as a criterion in determining method or field uniqueness. This feature can allow up to 15% more redundancy in method and field renames. In addition, since overloading on method return type or field type is typically not allowed in source languages (including C# and VB), this further hinders decompilers.

2.3.1.2 Control Flow

Traditional control flow obfuscation introduces false conditional statements and other misleading constructs in order to confuse and break decompilers. This process synthesizes branching, conditional, and iterative constructs that produce valid forward (executable) logic, but yield non-deterministic semantic results when decompilation is attempted. Control Flow obfuscation produces spaghetti logic that can be very difficult for a cracker to analyze.

Dotfuscator employs advanced control flow obfuscation. In addition to adding code constructs, Dotfuscator works by destroying the code patterns that decompilers use to recreate source code. The end result is code that is semantically equivalent to the original but contains no clues as to how the code was originally written. Even if highly advanced decompilers are developed, their output will be guesswork.

Consider the following example:

Original Source Code Before Obfuscation © 2001, Microsoft Corporation (Snippet from WordCount.cs C# example code)

```
public int CompareTo(Object o) {
    int n = occurrences – ((WordOccurrence)o).occurrences;
    if (n == 0) {
        n = String.Compare(word, ((WordOccurrence)o).word);
    }
    return(n);
}
```

Reverse-Engineered Source Code After Control Flow Obfuscation By Dotfuscator

```
public virtual int _a(Object A_0) {
    int local0;
    int local1;
    local0 = this.a – (c) A_0.a;
    if (local0 != 0) goto i0;
    goto i1;
    while (true) {
       return local1;
       i0: local1 = local0;
    }
    i1: local0 = System.String.Compare(this.b, (c) A_0.b);
    goto i0;
}
```

2.3.1.3 String Encryption

Dotfuscator allows you to hide user strings that are present in your assembly. A common attacker technique is to locate critical code sections by looking for string references inside the binary. For example, if your application is time locked, it may display a message when the timeout expires. Attackers search for this message inside the disassembled or decompiled output and chances are, when they find it, they will be very close to your sensitive time lock algorithm.

Dotfuscator addresses this problem by allowing you to encrypt strings in these sensitive parts of your application, providing an effective barrier against this type of attack.

Since string encryption incurs a slight runtime penalty no string encryption is performed except on the parts of the application that you specify.

2.3.1.4 Pruning

Smaller applications download faster, install faster, load faster and run faster. Dotfuscator's pruning feature statically analyzes your code to find the unused types, methods, and fields, and removes them. Dotfuscator also removes debug information and non-essential metadata from a MSIL file as it processes it, making the application smaller and reducing the data available to an attacker.

The static analysis works by traversing your code, starting at a set of methods called “triggers," or entry points. In general, any method that you expect external applications to call must be defined as a trigger. For example, in a simple standalone application, the **`Main`** method would be defined as a trigger. An assembly can have more than one trigger defined for it.

> Note that turning on library mode for an assembly causes Dotfuscator to treat all visible types and members as entry points, automatically.

As Dotfuscator traverses each trigger method’s code, it notes which fields, methods, and types are being used. It then analyses all the called methods in a similar manner. The process continues until all called methods have been analyzed. Upon completion, Dotfuscator is able to determine a minimum set of types and their members necessary for the application to run. Only these types are included in the output assembly.

Dotfuscator generates a removal report in XML format that lists all input assemblies and how each was pruned. Each assembly listing has a listing of types and their members (methods, fields, properties, etc.) along with an attribute indicating whether the item was pruned or not. The report also describes how managed resources attached to each assembly were pruned. At the end, the report provides a statistics section regarding the overall effectiveness of pruning.

2.3.1.5 Tamper Detection and Defense

Dotfuscator injects code that verifies your application’s integrity at runtime. If it detects tampering, it can shut down the application, invoke random crashes (to disguise that the crash was the result of a tamper check), or perform any other custom action. A tamper message can be sent to a PreEmptive Analytics endpoint, including PreEmptive Analytics Runtime Intelligence Service, to indicate that tampering was detected.

2.3.1.6 Shelf Life

Shelf Life is an application inventory management function that allows you to embed expiration, or de-activation, and notification logic into an application. Dotfuscator injects code that reacts to application expiration by exiting the application and/or sending a PreEmptive Analytics message. This feature is particularly helpful with beta applications. Users can schedule an application's expiration/de-activation for a specific date and optionally issue warnings to users that the application will expire/de-activate in a specific number of days.

DMA Shelf Life is unavailable in Dotfuscator for Marketplace Apps.

If you wish to provide the option of extending the shelf life, you may do so by writing code to provide updated expiration information to the shelf life engine by specifying a Shelf Life Token Source (e.g. an external configuration file). This provides a mechanism to extend warning and expiration dates without requiring re-instrumentation and redistribution of binaries.

Users must first obtain a *Shelf Life Activation Key (SLAK)* to use the Shelf Life feature. The key is issued by PreEmptive and provided to Dotfuscator by the user during shelf life configuration.

Supported .NET Application Types

Dotfuscator can perform Shelf Life Notification processing for all .NET assemblies except for the following:

- Managed C++ input assemblies containing native and managed code.
- Multi-module input assemblies.
- Silverlight assemblies.
- Windows Phone assemblies.
- WinRT assemblies.

2.3.1.7 Watermarking

Watermarking helps track unauthorized copies of your software back to the source by embedding data such as copyright information or unique identification numbers into a.NET application without impacting its runtime behavior. Dotfuscator's watermarking algorithm does not increase the size of your application, nor does it introduce extra metadata that could break your application.

2.3.2 Building and Debugging Obfuscated Applications

2.3.2.1 Linking

Dotfuscator can combine multiple input assemblies into one or more output assemblies. Assembly linking can be used to make your application even smaller, especially when used with renaming and pruning, and can simplify deployment scenarios.

For example, if you have input assemblies **A, B, C, D**, and **E**, you can link assemblies **A, B,** and **C** and name the result **F.** At the same time, you can also link **D** and **E** and name the result **G**. The only rule is that you can't link the same input assembly into multiple output assemblies.

The linking feature is fully integrated with the rest of Dotfuscator, so in one step, you can obfuscate, remove unused types, methods, and fields, and link the result into one assembly.

If you have other input assemblies that reference the assemblies you are linking together, Dotfuscator will transparently update all the assembly and type references so the output assemblies will correctly work together.

Linking is not supported for Managed C++ assemblies.

2.3.2.2 Incremental Obfuscation

Incremental obfuscation allows you to keep a consistent naming scheme across Dotfuscator runs for your application. By referencing a name mapping file, types and members can be consistently renamed over time. Consistent renaming is desirable in multiple scenarios, including the redistribution of a subset of files that constitute a dependant group, and sequential obfuscation of assemblies in a resource constrained environment. Appropriate utilization of this feature offers the additional benefit of more rapid obfuscation, when only some of the assemblies in a project need to be redistributed.

Consider a scenario where you have used Dotfuscator on your application and distributed that application to your customers. Now you would like to make changes to one of the assemblies and provide it as an update. A naive re-execution of Dotfuscator upon your application would likely rename your legacy classes and methods in a different way, forcing you to redistribute the entire application to your customers. Dotfuscator’s incremental obfuscation allows you to keep the same names so you can release the changed assembly.

Incremental obfuscation is useful in sequential build scenarios by allowing the obfuscation of large projects to be broken down into smaller, more manageable groups of assemblies. A hypothetical project consisting of three files, **`A.exe`**, **`B.dll`**, and **`C.dll`** where **`A`** references **`B`**, and **`B`** references **`C`** could be built as follows: **`C.dll`** could be obfuscated initially, then **`B.dll`** could be incrementally obfuscated using the map file from **`C.dll`**, and finally **`A.exe`** could be obfuscated using the map file from **`B.dll`**.

Incremental obfuscation requires an input mapping file containing the names that need to be reused. The format is the same as the output mapping file that Dotfuscator produces after every run. A best practice is to save a copy of the output mapping file in a safe place (e.g. in version control) for every released build of your application. The file can then be used as an input mapping file if an incremental update should ever be necessary.

When performing a run using incremental obfuscation, Dotfuscator must have access to all the application’s assemblies, although it is not required that all the assemblies be included in the project. They only need to be discoverable by the same probing rules used by Dotfuscator to locate referenced assemblies.

2.3.2.3 Debugging Obfuscated Code

One major drawback of obfuscation is that the task of maintaining and troubleshooting an obfuscated application becomes more difficult. In a well obfuscated application, the names of all types, methods, and fields are changed from their carefully thought out, informative names into meaningless names. This makes using a debugger more difficult, and impacts the usefulness of stack traces sent in from the field.

To solve the debugging problem, Dotfuscator has the ability to output debugging symbol files for obfuscated application that correspond as closely as possible to the original symbol files output by the compiler. Using these files, developers can use a debugger to step through an obfuscated assembly and view the original source code.

To solve the stack trace problem, PreEmptive Solutions provides **Lucidator**, a standalone tool that translates and decodes stack traces emitted by programs obfuscated with Dotfuscator. Lucidator works by automatically decoding obfuscated stack traces using the renaming map file. Given the obfuscated stack trace, Lucidator replaces the obfuscated names with the original names and displays the results. This same translation ability is built into the Dotfuscator GUI, as well.

Lucidator does not need to be run on the same machine on which Dotfuscator is installed, as long as it has access to the appropriate map file.

2.3.3 Advanced Topics

2.3.3.1 Smart Obfuscation

Smart Obfuscation is an ongoing effort to identify and apply obfuscation rules automatically for known API usage patterns and application types.

The current implementation recognizes applications and libraries that use some common technologies and patterns, such as:

- Windows Presentation Foundation (Beginning with 4.7.1000 these exclusions are disabled if the Transform XAML option is enabled for the Assembly)
- Windows Communication Foundation
- Windows Workflow
- Windows Cardspace
- Data-bound Windows Forms controls
- Enumerated type values used as strings
- Late calls in VB applications
- Custom Serialization
- Web Services
- Silverlight (Beginning with 4.8.1000 these exclusions are disabled if the Transform XAML option is enabled for the Assembly)

For these application types, Dotfuscator’s renamer and pruner can make a best effort to identify cases where renaming or removing elements will break the output application. It then automatically prevents renaming or removal without additional user configuration.

The *Smart Obfuscation* rules use static analysis to determine what elements should be excluded from renaming or used as removal entry points. When such an item is discovered, the rule issues a notification that displays in the *Smart Obfuscation* tab. Sometimes a rule can recognize that an action needs to be taken, but cannot determine what specific action to take because static analysis does not yield enough information. When this happens, the rule issues a warning that displays in the *Smart Obfuscation* tab.

Dotfuscator allows you to turn Smart Obfuscation off. Smart Obfuscation is turned on by default, and in most cases should be left on. It can be turned off in cases where the user believes that more aggressive obfuscation will not hurt the application. There are several ways to turn it off:

- Changing the **`<option>`** setting in the **`<smartobfuscation>`** section to **`disable`**.
- From the Project Properties dialog in the Visual Studio UI.
- From the Settings Tab in the standalone UI.

Dotfuscator allows you to control the verbosity of Smart Obfuscation reporting. You can choose to report all actions and warnings, warnings only, or suppress reporting altogether. There are several ways to set the reporting level:

- Setting the **`verbosity`** attribute on the smartobfuscationreport element in the configuration file.
- From the Project Properties dialog in the Visual Studio UI.
- From the Settings Tab in the standalone UI.

Dotfuscator allows you to save the Smart Obfuscation report to a file in addition to displaying it. You may instruct Dotfuscator to save the report to a file by specifying a file path in any of the following places:

- A **`file`** element within the smartobfuscationreport element in the configuration file.
- From the Project Properties dialog in the Visual Studio UI.
- From the Settings Tab in the standalone UI.

Dotfuscator automatically renames an existing smart obfuscation report with the same name before overwriting it with a new version. If you do not want this behavior, there are several ways to instruct Dotfuscator not to rename existing removal reports before overwriting:

- Setting the **`overwrite`** attribute on the smartobfuscationreport element in the configuration file to **`true`**.
- From the Project Properties dialog in the Visual Studio UI.
- From the Settings Tab in the standalone UI.

Smart Obfuscation Report

Dotfuscator generates a Smart Obfuscation report in XML format that lists all items flagged by the Smart Obfuscation process. Keep in mind that the contents of the report reflect the Smart Obfuscation reporting verbosity setting - if the verbosity is set to *Warnings Only* or *None*, items flagged by Smart Obfuscation may be omitted from the report. Each entry represents one item that was flagged by Smart Obfuscation, and has a description of what the item was and why it was excluded, along with an attribute indicating whether the flag action was a warning or a notification.

The elements of the Smart Obfuscation report are similar to those in the map file and removal report. A few things are noteworthy:

- The report includes the name of the rule that flagged the item, the identity of the item, the action taken, and the reason for that action.
- The identity of the flagged item is specified by the type of item (type, method, field, property, etc.) and its full signature.
- In type names, nested class names are separated from the parent using the "**`/`**" character.
- Constructors are named **`.ctor`**, while static constructors (a.k.a. static initializers, class constructors, etc) are named **`.cctor`**.

2.3.3.2 P/Invoke Methods

P/Invoke methods (*i.e.* native platform methods) are automatically not renamed if Dotfuscator detects that the name is used to find the corresponding native function. P/Invoke methods that are mapped to native functions by an alias or by ordinal can be renamed.

2.3.3.3 Managed C++ and IJW (It Just Works) Thunking

Dotfuscator can process assemblies containing managed and unmanaged (native) code, such as those created by the Managed C++ compiler. Dotfuscator performs renaming and metadata removal on mixed code modules; however, string encryption, control flow obfuscation, and pruning are automatically disabled. These features are still enabled on other "pure managed" input modules included in the run.

2.3.3.4 Dotfuscating Assemblies with Managed Resources

Managed resources may be embedded inside a module (internal) or may be in files external to the module. Often, part of the name of the managed resource is a type name (see the .NET Framework documentation for more information about the "hub and spoke model" for lookup of managed resources.).

When the type name is renamed, Dotfuscator attempts to locate and rename the corresponding managed resource. If the resource is internal to the assembly, this is automatic. If the resource is embedded inside an external file, then the file must be in the same directory as the referencing module. If the resource is embedded inside another assembly, then that assembly must be one of the input assemblies.

2.3.3.5 Dotfuscating Assemblies with Satellite DLLs

Localized applications can be seamlessly obfuscated along with their satellite resource DLLs. Dotfuscator automatically discovers these DLLs using the same rules that the runtime uses and automatically adds them as inputs to the obfuscation process. You do not need to explicitly specify them as inputs.

Your localized resources contained in the satellite DLLs will be renamed in synch with your culture neutral resources in the main assembly.

2.3.3.6 Dotfuscating Multi-module Assemblies

A .NET assembly may be made up of multiple modules (*i.e.* files on disk). Usually an assembly is made up of one module, and this is the scenario that most tools, such as Visual Studio, support. Occasionally it is desirable to create assemblies made up of more than one module. Dotfuscator supports this scenario. Note that Dotfuscating a multi-module assembly is not the same as Dotfuscating multiple input assemblies.

To Dotfuscate a multi-module assembly, only the prime module needs to be listed as an input assembly. The non-prime modules are searched for in the same directory as the prime module.

In the prime module’s assembly manifest, Dotfuscator automatically updates the hash values of the other modules.

2.3.3.7 Dotfuscating Strong Named Assemblies

Strong named assemblies are digitally signed. This allows the runtime to determine if an assembly has been altered after signing. The signature is an SHA1 hash signed with the private key of an RSA public/private key pair. Both the signature and the public key are embedded in the assembly’s metadata.

Since Dotfuscator modifies the assembly, it is essential that signing occur *after* running the assembly through Dotfuscator.

Dotfuscator handles this step as part of the obfuscation process.

Automatically Re-signing after Obfuscation

Dotfuscator automatically re-signs strongly named assemblies after obfuscation, eliminating the need for manual steps after obfuscation. Dotfuscator both re-signs your already signed assemblies, and completes the signing process on delay signed assemblies.

Re-signing Strongly Named Assemblies

As part of the build process, Dotfuscator re-signs assemblies that are already strongly named. You can tell Dotfuscator explicitly where to find the public/private key pair, or you can rely on a location specified by custom attributes on the input assembly (e.g. **System.Reflection.AssemblyKeyFileAttribute**).

The following example shows an XML configuration file fragment that sets up resigning with an explicit key file. Any key file specified via custom attribute is not used.

Re-signing with Explicit Key File

```
  <signing>
    <resign>
      <option>dontuseattributes</option>
      <key>
        <file dir="c:\temp" name="key.snk" />
      </key>
    </resign>
    ...
  </signing>
```

Finishing Signing Delay Signed Assemblies

If an input assembly is delay signed, Dotfuscator can finish the signing process. Tell Dotfuscator where to locate the private key required to complete the signing.

The following example shows an XML configuration file fragment that sets up delay signing with an explicit key file.

Delay Signing

```
<signing>
  ...
    <delaysign>
      <key>
        <file dir="c:\temp" name="key.snk" />
      </key>
    </delaysign>
</signing>
```

2.3.3.8 Authenticode Signing Assemblies

Authenticode signed assemblies are digitally signed by a code signing certificate issued by a trusted root certificate authority. This allows the operating system and runtime to determine the publisher of an application and to determine if the assembly has been altered after being signed. The signature is a hash encrypted with the private key of a code signing certificate. Both the signature and the public key are embedded in the assembly's metadata.

Since Dotfuscator modifies the assembly, it is essential that Authenticode signing occur *after* running the assembly through Dotfuscator.

Dotfuscator handles this step as part of the obfuscation process.

Automatic Authenticode Signing after Obfuscation

As part of the build process, Dotfuscator performs Authenticode signing of output assemblies. You must tell Dotfuscator explicitly where to find the code signing certificate store as a PFX container and optionally the password for the container.

Dotfuscator provides the ability for you to specify the URL of an Authenticode timestamp service when performing Authenticode signing. This URL will be accessed during Dotfuscator's signing process, and will provide additional data which will allow your assemblies' Authenticode signatures to remain valid after your code-signing certificate has expired.

The following example shows an XML configuration file fragment that performs Authenticode signing.

Authenticode Digital Signing

```
  <digitalsigning>
    <pfx password="secret123">
      <file dir="C:\test" name="authenticode.pfx" />
    </pfx>
  </digitalsigning>
```

2.3.3.9 Dotfuscating 64-Bit Assemblies

Dotfuscator can transparently obfuscate managed assemblies written explicitly for specific CPU architectures, including 64-bit architectures.

Dotfuscator itself is a managed application and can run on 32-bit and 64-bit versions of Windows.

2.3.3.10 Reflection and Dynamic Class Loading

Reflection and dynamic class loading are extremely powerful tools in the .NET architecture. This level of runtime program customization prevents Dotfuscator from infallibly determining whether it is safe to rename all types loaded into a given program.

Consider the following (C#) code fragment:

C# Code Fragment:

```
public object GetNewType() {
   Type type = Type.GetType( GetUserInputString(), true );
   object newInstance = Activator.CreateInstance( type );
   return newInstance;
}
```

This code loads a type by name and dynamically instantiates it. In addition, the name is coming from a string input by the user!

There is no way for Dotfuscator to predict which type names the user will enter. The solution is to configure Dotfuscator to exclude the names of all potentially loadable types. Note that method and field renaming can still be performed. This is where manual user configuration plays an important role.

Often the situation is less serious. Consider a slight variation:

C# Code Fragment Variation:

```
public MyInterface GetNewType() {
   Type type = Type.GetType( GetUserInputString(), true );
   object newInstance = Activator.CreateInstance( type );
   return newInstance as MyInterface;
}
```

Now it is immediately obvious that only a subset of types need to be excluded: those implementing MyInterface.

2.3.3.11 Declarative Obfuscation using Custom Attributes

The .NET Framework provides two custom attributes designed to make it easy to automatically obfuscate assemblies without having to set up configuration files. This section outlines how you can use these attributes with Dotfuscator. It is assumed that you are familiar with custom attributes and how to apply them in your development language.

If you are using an earlier version of the .NET Framework, Dotfuscator ships with a DLL containing compatible attributes. To use these, add the **`PreEmptive.ObfuscationAttributes.dll`** as a reference when building your project. When referencing the compatible attributes in your source code, replace the **`System.Reflection`** namespace with **`PreEmptive.Dotfuscator.ObfuscationAttributes`**.

System.Reflection.ObfuscateAssemblyAttribute

This attribute is used at the assembly level to tell Dotfuscator how to obfuscate the assembly as a whole. Setting the AssemblyIsPrivate property to false tells Dotfuscator to run the assembly in library mode. If you set it to true, Dotfuscator will not run the assembly in library mode and will rename as much as possible, including public types and members.

System.Reflection.ObfuscationAttribute

This attribute is used on types and their members and tells Dotfuscator how to obfuscate the item.

Feature Property

This string property has a default value of "**`all`**". This property is provided so that you can configure multiple obfuscation transforms independently by tagging an item with multiple **`ObfuscationAttributes`**, each with a different feature string.

Dotfuscator maps the "**`default`**" and "**`all`**" feature strings to "**`renaming`**".

Here is a list of other feature strings that Dotfuscator understands.

Feature String	Action
renaming	attribute configures renaming
controlflow	attribute configures control flow obfuscation.

stringencryption	attribute configures string encryption
trigger	attribute configures pruning by marking the annotated item as an entry point
conditionalinclude	attribute configures pruning by conditionally including the annotated item

If necessary, you can map other feature strings to **`renaming`** using the **Feature Map Strings** property sheet on the *Setup* Tab.

Dotfuscator ignores attributes with feature strings that it does not understand.

Exclude Property

This Boolean property has a default value of True. When True, it indicates that the tagged item should be excluded from the transforms implied by the Feature property. When False, it indicates that the tagged item should be included.

The current version of Dotfuscator supports one value of the Exclude property for any given transform. Dotfuscator will ignore rules that have unsupported Exclude values. The following list summarizes.

Feature String	Supported Exclude Value
renaming	True
controlflow	True
stringencryption	False
trigger	False
conditionalinclude	False

ApplyToMembers Property

This Boolean property has a default value of True. When the attribute is applied to an assembly or a type, a True value indicates that the operation should be applied to all the members, including nested types, of selected types. If false, the operation is applied to types only and not their members or nested types.

Enabling or Disabling Declarative Obfuscation

Dotfuscator allows you to switch Declarative Obfuscation on or off for all input assemblies. If not enabled, Dotfuscator ignores obfuscation related custom attributes. You can also switch it off for specific assemblies.

Stripping Declarative Obfuscation Attributes

Dotfuscator can strip out the obfuscation attributes when processing is complete, so output assemblies will not contain clues about how it was obfuscated. Both of the declarative obfuscation attributes include a Boolean "**`StripAfterObfuscation`**" property whose default value is true

Dotfuscator also has configuration settings that interact with the value of the **`StripAfterObfuscation`** property at obfuscation time.

The settings that effect declarative obfuscation attribute stripping and how they interact are summarized in the below table.

Dotfuscator is Honoring Attributes	Dotfuscator is Stripping Attributes	Attribute's StripAfterObfuscation Property	Result
Yes	Yes	True or False	Strip Attribute
Yes	No	True	Strip Attribute
Yes	No	False	Keep Attribute
No	Yes	True or False	Strip Attribute
No	No	True or False	Keep Attribute

Using Feature Map Strings

Dotfuscator allows you to map values contained in an Obfuscation Attribute's Feature property to feature strings that Dotfuscator understands.

For example, you can annotate your application with obfuscation attributes that reference a feature called "**`testmode`**". Dotfuscator, by default, does not understand this feature string; therefore it ignores the attributes. Later, if you want Dotfuscator to use these attributes to configure renaming and controlflow obfuscation, then map the feature string "**`testmode`**" to Dotfuscator's built in "**`renaming`**" and "**`controlflow`**" strings.

2.3.3.12 Build Events

Dotfuscator allows you to specify programs that run before and after its build sequence.

Pre Build Event

In its build process, Dotfuscator executes the program specified by the pre build event before it does anything else with your input assemblies.

Post Build Event

Dotfuscator executes the program specified by the post build event at the very end of its build process. You can tell Dotfuscator to execute the program only when the build succeeds, only when the build fails, or all the time. In addition, you can tell Dotfuscator to run the program once for each output module.

Build Event Properties

The Dotfuscator build engine exposes several properties that you can use when configuring build events:

Property Name	Description
dotf.destination	Path to the destination directory.

dotf.inputmap.xml	Full path and filename to the input map file if specified.
dotf.outputmap.xml	Full path and filename to the output map file if specified.
dotf.removal.xml	Full path and filename to the XML removal file if specified.
dotf.config.file	Full path to the current configuration file.
dotf.current.out.module	Full path to the current output module. Used in the post build event when the program is called for each output module.
dotf.current.in.module	Full path to the current input module. Used in the post build event when the program is called for each output module.

You can also reference external properties (environment variables or properties passed on the command line using the **`-p`** option) and user defined project properties in your build events.

2.3.3.13 Friend Assemblies

The .NET Framework has the concept of *friend assemblies*, where an assembly may declare that its internal type definitions are visible to specified other assemblies. This is done using the **`System.Runtime.CompilerServices.InternalsVisibleToAttribute`**. Dotfuscator detects the use of this attribute and modifies its renaming and pruning rules as described below.

Assume **A** and **B** are two assemblies where assembly **B** references **A**, and **A** is marked with **`InternalsVisibleTo ( B ).`**

There are a few cases of interest:

1. **A** and **B** are both input assemblies and Dotfuscator is in library mode. If **A** has no other external, non-input assembly friends, Dotfuscator safely mangles internal names and fixes up the references in assembly **B**. If **A** does have other external friends, then internal names are preserved.
2. **A** and **B** are both input assemblies and Dotfuscator is not in library mode. Internal names in **A** are, by default, mangled and references in **B** are fixed up, regardless of the existence of other external friend assemblies. As usual in Dotfuscator, names and groups of names can be preserved via manual configuration.
3. **A** is an input assembly, **B** is not, and Dotfuscator is in library mode. Internal names in **A** are not mangled in order to not break potential references in **B**.
4. **A** is an input assembly, **B** is not, and Dotfuscator is not in library mode. Internal names are mangled, potential references in **B** are not fixed up since it is not an input assembly. As usual in Dotfuscator, names and groups of names can be preserved via manual configuration. This case would require manual configuration if **B** actually does reference **A's** internals.

2.3.3.14 Finding External Tools

Dotfuscator uses ildasm and ilasm to process the input assemblies. Ildasm is the MSIL disassembler that ships with the .NET Framework SDK. Ilasm is the MSIL assembler that ships with the .NET Framework Redistributable.

On systems with .NET 1.1 or below, Dotfuscator attempts to match each input assembly with the toolset that ships with the version of the .NET Framework that it was compiled with. If Dotfuscator cannot find the version appropriate toolset for an input assembly, it uses a later version if present. It never uses an older version.

On systems with .NET 2.0 and above, Dotfuscator will use the latest tools even for .NET 1.x assemblies. When building, Dotfuscator passes the appropriate command line arguments to ilasm to ensure that the output assemblies target their correct framework versions.

By default, Dotfuscator searches for these external tools using the following algorithm:

- Determine the version of the .NET Framework that the input assembly was compiled on.
- Look at user specified properties that override the default locations of the tools. You can do this in a tool and version specific way. The following table summarizes by example:

Property	Value	
ILASM_v1.0.3705	C:\tools\ilasm.exe	.NET v1.0.3705 assemblies will use this version of ilasm.
ILDASM_v2.0.50215	C:\tools\ildasm.exe	.NET v2.0.50215 assemblies will use this version of ildasm.
ILDASM_v1.1	C:\tools\ildasm.exe	.NET v1.1.xxxx assemblies will use this version of ilasm.

Note: These properties are case sensitive.

- Search the .NET Framework and .NET Framework SDK directories corresponding to the .NET Framework version determined in step 1.
- Search the .NET Framework and .NET Framework SDK directories corresponding to later versions of the .NET Framework determined in the first step.

If Dotfuscator cannot find one or both of these programs, it issues an error.

Dotfuscator uses the strong naming tool (**`sn.exe`**) to automatically resign your strong named assemblies. This tool also ships with the .NET Framework SDK and Dotfuscator searches for it in the same directory as ildasm.

2.4 Understanding Instrumentation with Dotfuscator

In addition to obfuscation, Dotfuscator also provides features that inject code into an application. Some of that code is used to enhance the protection provided by obfuscation (i.e. Tamper Notification and Shelf Life), but Dotfuscator can also inject code that sends messages back to you about how the application is used in production. This powerful capability lets you deeply understand your application usage, which can help you make better decisions about your software priorities.

This injected code is called **Instrumentation**, and is most-often used with one of the **PreEmptive Analytics** products: Runtime Intelligence Service or PreEmptive Analytics for Team Foundation Server.

Runtime Intelligence Service

Runtime Intelligence Service is a hosted portal that gives application authors insight into how their applications are used. Dotfuscator can inject the instrumentation code that will send data to the Runtime Intelligence Service. This includes:

- Tamper Notification
- Shelf Life
- Exception Tracking and Application Analytics, as detailed below

Dotfuscator can instrument an application such that a message is sent when the application starts and stops, a feature is used, or an exception occurs. The instrumentation can also send back data about the runtime platform, unique users and installations, and performance of the application. The Runtime Intelligence Service aggregates this lifecycle data from the application and exposes it through the Runtime Intelligence Portal, available to Runtime Intelligence Service subscribers. To use this functionality, you must be a Runtime Intelligence Service subscriber.

PreEmptive Analytics for Team Foundation Server (TFS)

PreEmptive Analytics for Team Foundation Server aggregates and analyzes exceptions and automatically creates Visual Studio / TFS work items based entirely upon rules and operational thresholds that you define. PreEmptive Analytics for TFS is designed specifically to help streamline feedback driven development, improve software quality and user experience, and increase development efficiency.

PreEmptive Analytics for TFS uses exception instrumentation. It does not require feature, platform, performance, etc. instrumentation.

Supported .NET Application Types

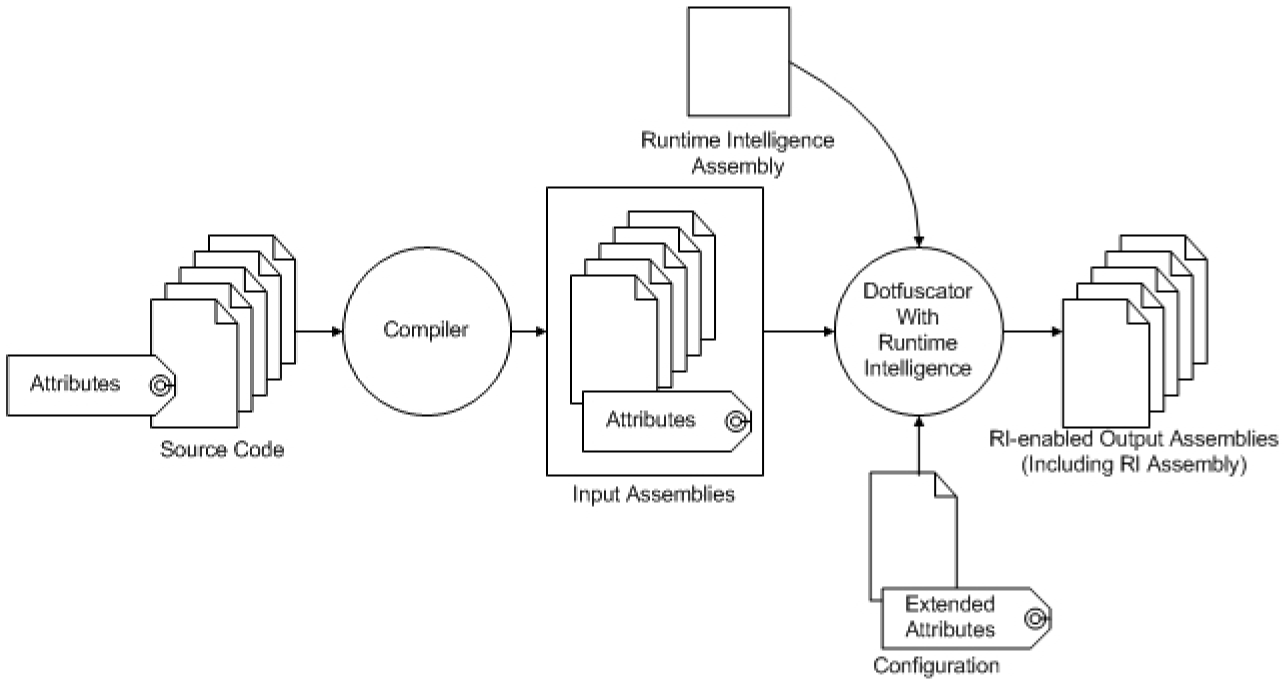
Dotfuscator can inject Instrumentation code for all .NET assemblies except for the following:

- Managed C++ input assemblies containing native and managed code.
- Multi-module input assemblies.
- Input assemblies that target .NET 1.0
- WinRT (coming soon)

2.4.1 Instrumentation Injection

The development workflow for adding Application Analytics to an application is similar to obfuscating an assembly. Both require configuring Dotfuscator to work with the specific assemblies, and both produce a set of output assemblies that match the input assemblies. The only difference is the specific configuration elements that are used. The output assemblies will contain all the run-time code necessary to collect and deliver the data.

The diagram below illustrates the workflow from the developer's point of view.


Runtime Intelligence Assembly
Attributes
Source Code
Compiler
Attributes
Input Assemblies
Dotfuscator With Runtime Intelligence
RI-enabled Output Assemblies (Including RI Assembly)
Extended Attributes
Configuration

PreEmptive Analytics Message Types

PreEmptive Analytics defines several message types:

- Application and Session Start
- Application and Session Stop
- Feature
- Performance Probe
- System Profile
- Tamper Detected
- Exception Detected

Application and Session **Start** and **Stop** messages (the *application lifecycle messages*) are intended to be sent when an *application* starts running and when it shuts down. The information contained in these messages is used to track application behavior and basic usage patterns. Extended usage and environment information is obtained by using the **Feature**, **Performance Probe**, or **System Profile** messages.

The data from these messages drive the Runtime Intelligence Portal’s dashboards. To have your application send these messages, you must:

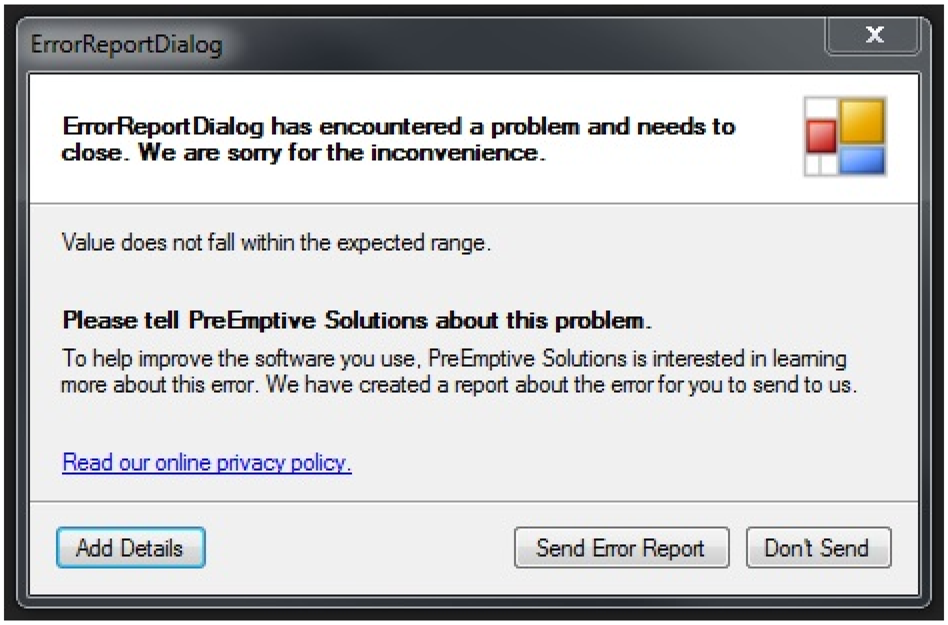
- Be a Runtime Intelligence Service subscriber (this gives you access to the dashboards and data in the portal).
- Configure your application with PreEmptive Analytics attributes, including Setup and Teardown. This can be done with attributes in your source code, or by configuring the attributes through the Dotfuscator GUI.
- Run your application through Dotfuscator with the **Send Analytics Messages** option turned on. See Configuring and Running Dotfuscator with Application Analytics.

See Example PreEmptive Analytics Enabled Application to see the contents of messages containing PreEmptive Analytics data.

2.4.2 Exception Tracking

Exception Tracking is a method for automatically detecting and responding to exceptions in the target application as they occur. Dotfuscator injects code that can detect caught, thrown, or unhandled exceptions. Once detected, the exception tracking code can collect details from the user and report the detected exception to a PreEmptive Analytics endpoint. A user can explicitly allow an exception report to be sent even if he or she has previously opted out of sending PreEmptive Analytics messages, and can provide comment and contact information to be sent along with the report. In addition, the developer can specify a custom action be taken when an exception is detected.

To facilitate the common use case of unhandled exception reporting, Dotfuscator can inject a pre-made Exception Report Dialog which provides a consistent user experience for reporting exceptions. Minimal configuration is needed to instruct Dotfuscator to track unhandled exceptions, display the exception report dialog which will obtain explicit user consent and collect optional comment and contact information from the user, and send the report to the configured PreEmptive Analytics endpoint.


ErrorReportDialog

ErrorReportDialog has encountered a problem and needs to close. We are sorry for the inconvenience.

Value does not fall within the expected range.

Please tell PreEmptive Solutions about this problem.

To help improve the software you use, PreEmptive Solutions is interested in learning more about this error. We have created a report about the error for you to send to us.

Read our online privacy policy.

2.4.3 Tamper Notification

Dotfuscator can instrument applications to detect if they have been tampered with and if so, optionally send a message to a PreEmptive Analytics Endpoint.

When run on a properly attributed .NET application, Dotfuscator processes the Tamper Notification attributes and instruments the application accordingly. The resulting output application will be ready to send Tamper Notifications to a PreEmptive Analytics Endpoint. The only differences between Tamper Notification and Application Analytics at this level are in the attributes.

Supported .NET Application Types

Dotfuscator can perform Tamper processing for all .NET assemblies except for the following:

- Managed C++ input assemblies containing native and managed code.
- Multi-module input assemblies.
- Input assemblies that target .NET 1.0.
- .NET Compact Framework assemblies.
- Silverlight assemblies.
- Windows Phone assemblies.
- WinRT assemblies.

2.5 Configuring Dotfuscator via the GUI

There are two user interfaces available with Dotfuscator:

- A standalone GUI
- A tightly-integrated set of components for Visual Studio

Both user interfaces allow you to configure all aspects of a dotfuscator project. The standalone GUI is generally preferable for larger projects or projects with complex inputs and outputs. The Visual Studio GUI is generally preferable for smaller projects that are typically built entirely within Visual Studio. Both user interfaces generate a Dotfuscator project file that is compatible with the other user interface, and that can be integrated into automated build scripts.

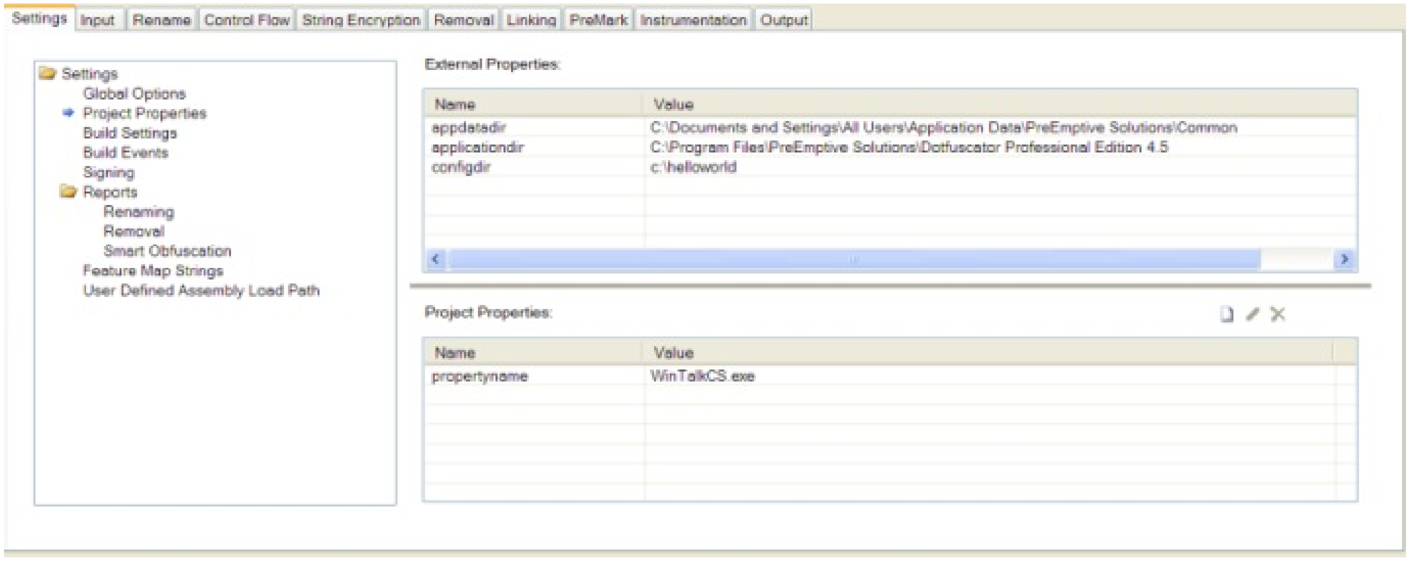
The first two topics in this section describe the components of the two user interfaces that are different from each other, while the remaining sections describe elements and features that are common to both.

2.5.1 The Standalone GUI

The Main Window

The main Dotfuscator window consists of four vertically arranged activity zones:

- At the top, the familiar Windows Menu Bar
- Below that menu, a toolbar appears for frequently accessed actions
- Below the toolbar, a tabbed section appears that organizes the specification and command activities for the Project
- At the bottom, a tabbed, scrollable console pane is provided for viewing output.



External Properties:

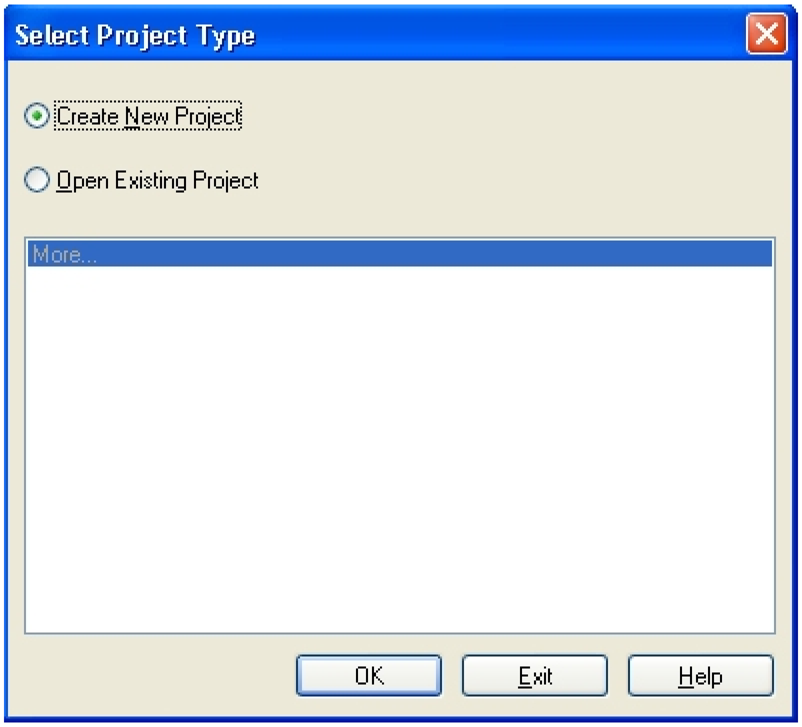
Name	Value
appdatadir	C:\Documents and Settings\All Users\Application Data\PreEmptive Solutions\Common
applicationdir	C:\Program Files\PreEmptive Solutions\Dotfuscator Professional Edition 4.5
configdir	c:\helloworld

Project Properties:

Name	Value
propertyname	WinTalkCS.exe

The Select Project Window

Start the user interface, without a project loaded, by invoking **`dotfuscator.exe`** from the installation directory, or by clicking on the icon created by the installer. When the *Select Project Type* window displays you can either create a new project, select an existing project from the most recently used list, or by selecting **More...** browse the file system for a project.

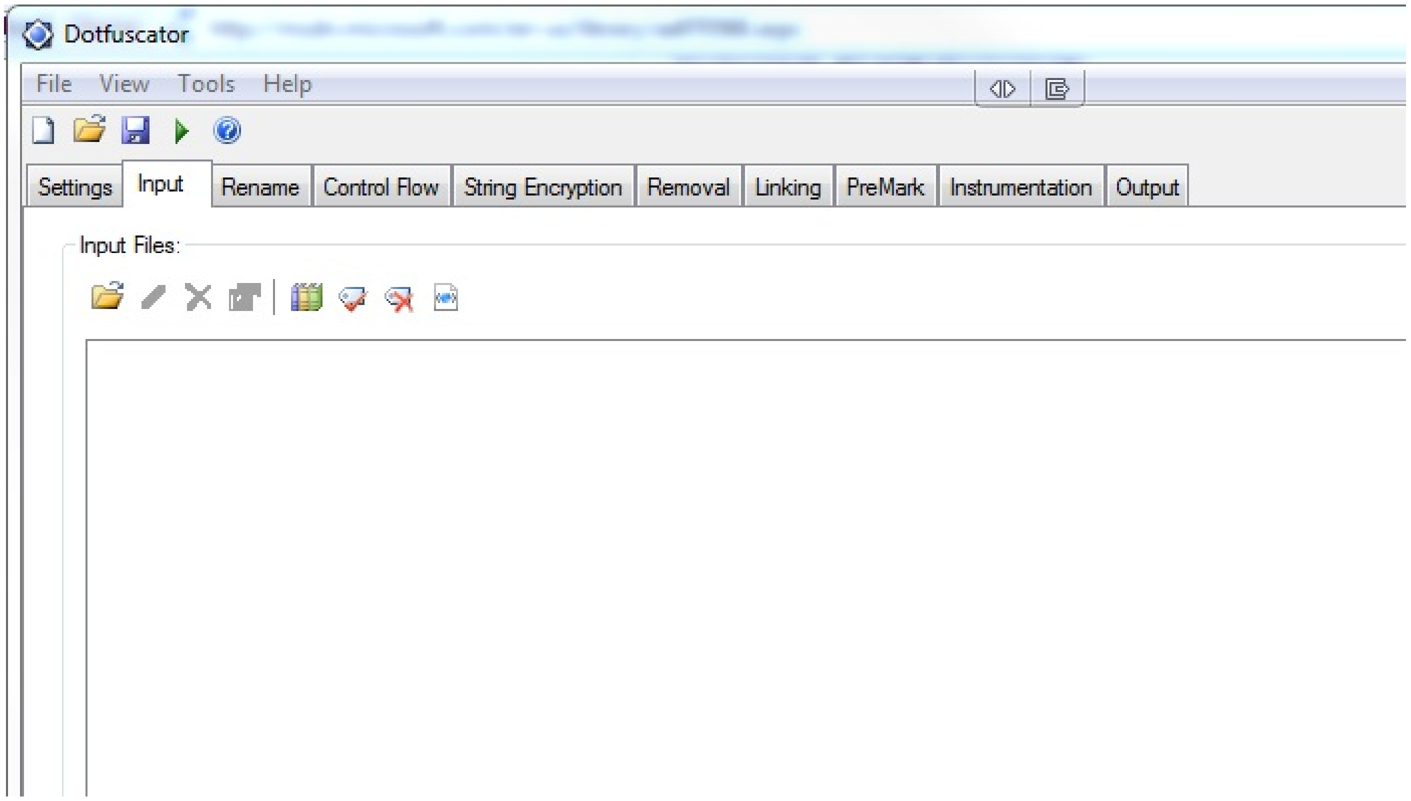


2.5.1.1 Working with Projects

To create a new project, you need to do three things:

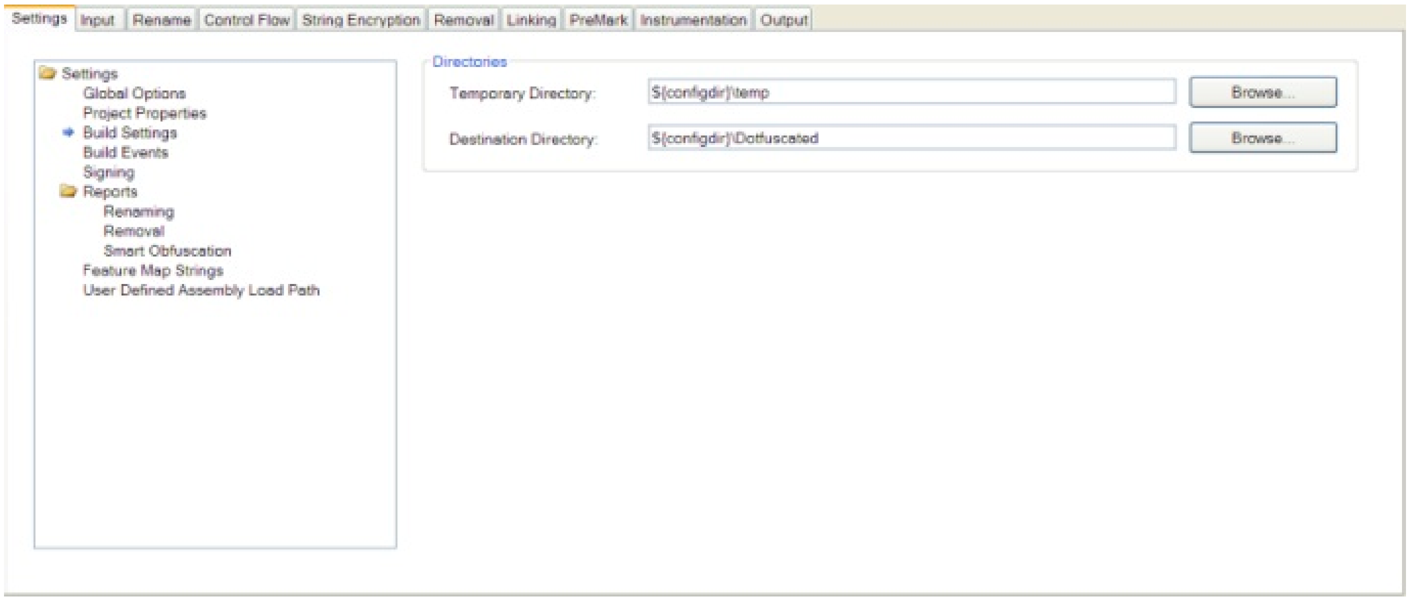
1: Select Inputs

From the *Input* Tab, you can use the toolbar to add an input package and/or assembly. From the *Add Input* dialog, you can type in the package or assembly’s directory and file name, type in a directory and file mask, or browse the file system for an input or folder.



2: Specify the Destination Directory

When you create a new project, the output directory is set by default to **${configdir}\Dotfuscated**. The **${configdir}** is a built-in property that is expanded to the folder that your project file is saved to.



Temporary Directory: ${configdir}\temp

Destination Directory: ${configdir}\Dotfuscated

Alternatively, you can browse your file system for the intended destination directory. The **Browse** button on the right of the *Destination Directory* field brings up the *Select Destination Directory* dialog that provides a directory navigation tree.

3: Save the Project Configuration File

You can save your project by either selecting **File > Save Project** or **File > Save Project As** from the menu or by clicking on the **Save Project** button on the toolbar. Navigate to your project directory, fill in your project configuration file name, click the **Save** button, and the project will be saved.

2.5.1.2 Working with Inputs

Adding Assemblies

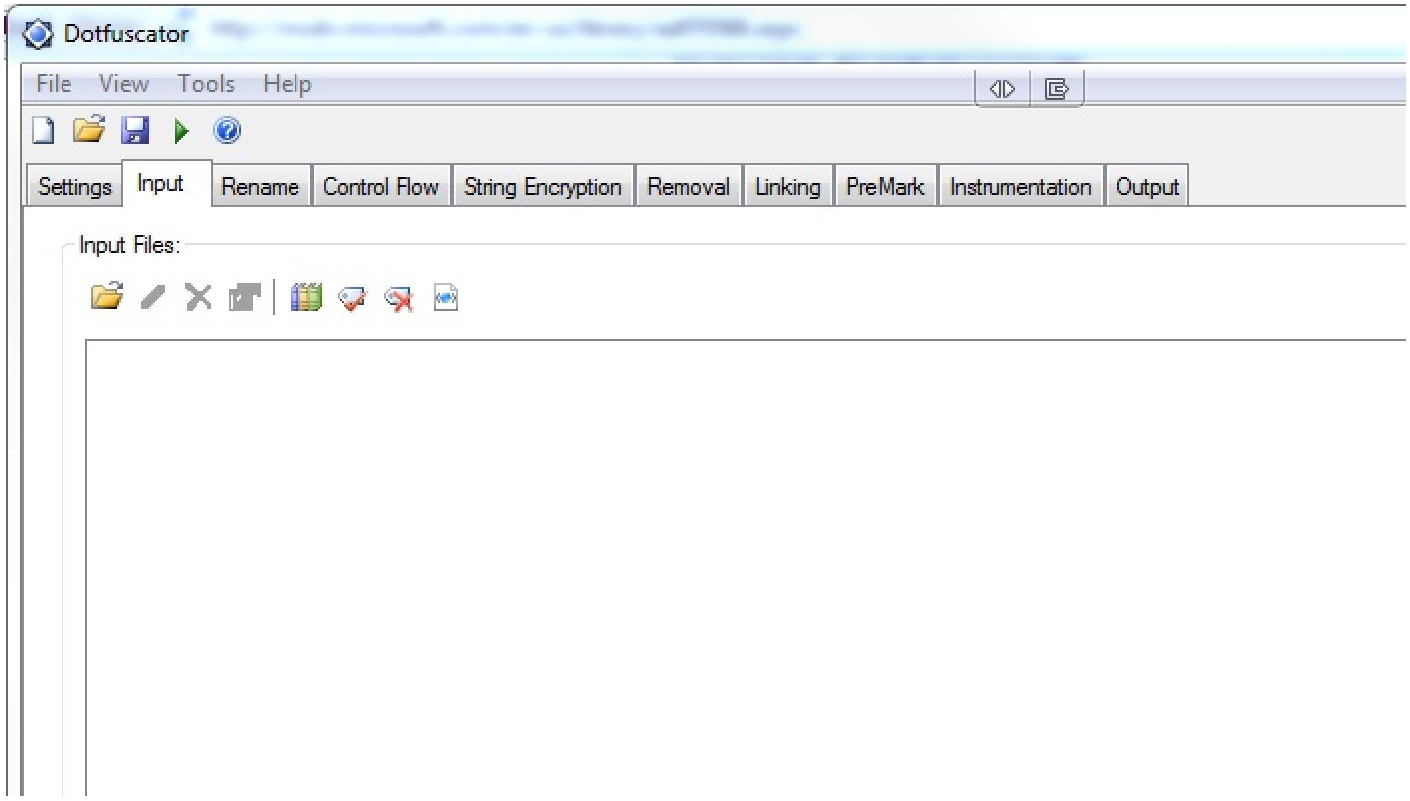
The *Input* tab is used to specify a combination of packages and/or assemblies for the project. With the **Add New Input** button on the toolbar, you can add a new package or assembly to your project. Clicking on the button brings up the *Add Input* dialog, where you can enter the package or assembly's directory and file name or file mask, or browse for it in the file system.

Using the *Browse* window, you can add multiple packages and/or assemblies by multi-selecting the ones you wish to add.

You may also drag and drop packages or assemblies into the Input Assemblies list.

Editing and Removing Inputs

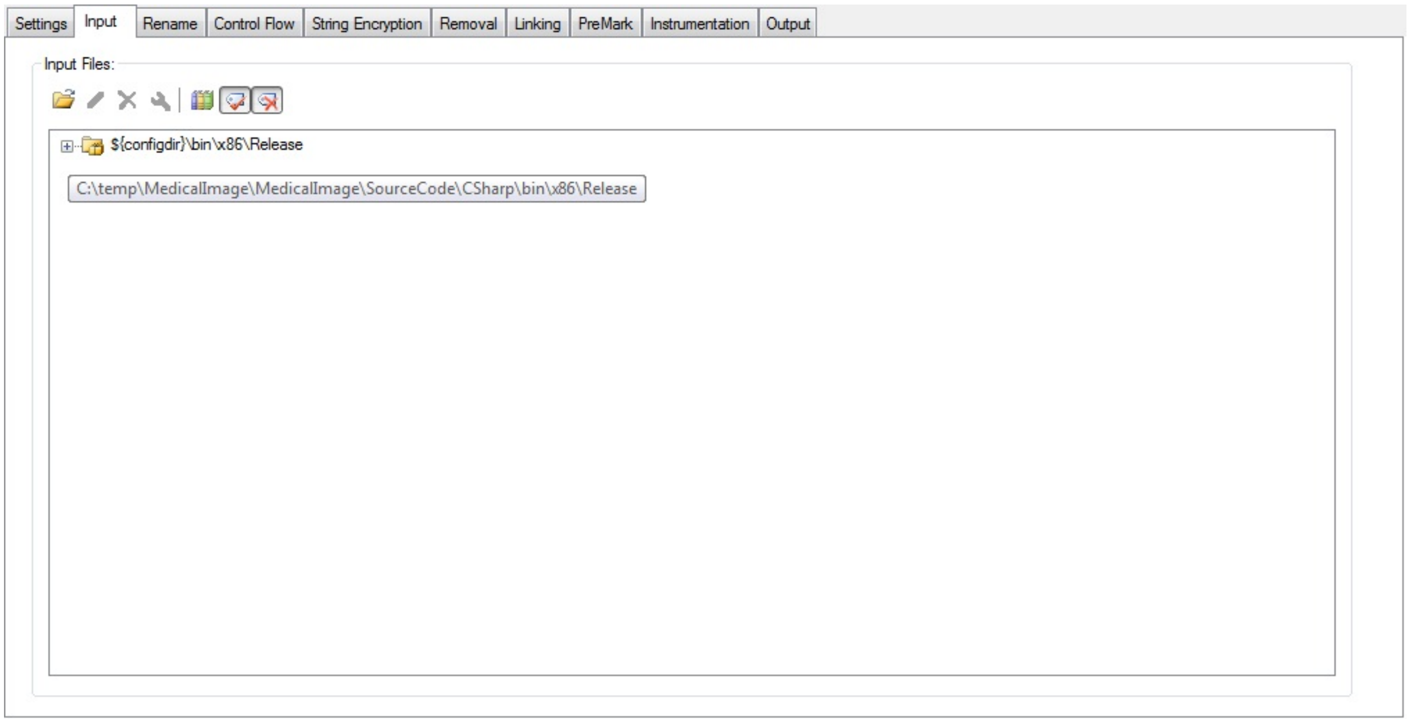
The **Edit** and **Remove** buttons are used to change or remove inputs from the project. To use, highlight an input on the list and click on the appropriate toolbar button. You can also delete an input by highlighting it and pressing the **Delete** key.



When editing an input path, the text you type can have a Project Property embedded in it. For example:

Project Property
`c:\${inputdir}\myapplication.exe`

Property substitution takes place based on the precedence rules specified in the section on Property List and Properties. You can view the actual, fully resolved value by placing the cursor on the item that has a Project Property embedded in it.



${configdir}\bin\x86\Release

C:\temp\MedicalImage\MedicalImage\SourceCode\CSharp\bin\x86\Release

Input Properties

Packages can have specific options. If the selected package has any available options the Input Properties button on the toolbar will be active.

Library Mode

Library mode can be toggled for all assemblies using the library button on the toolbar.

Alternatively, you can select library mode for specific assemblies by checking or un-checking the **Library** checkbox under the input assembly's entry in the list.

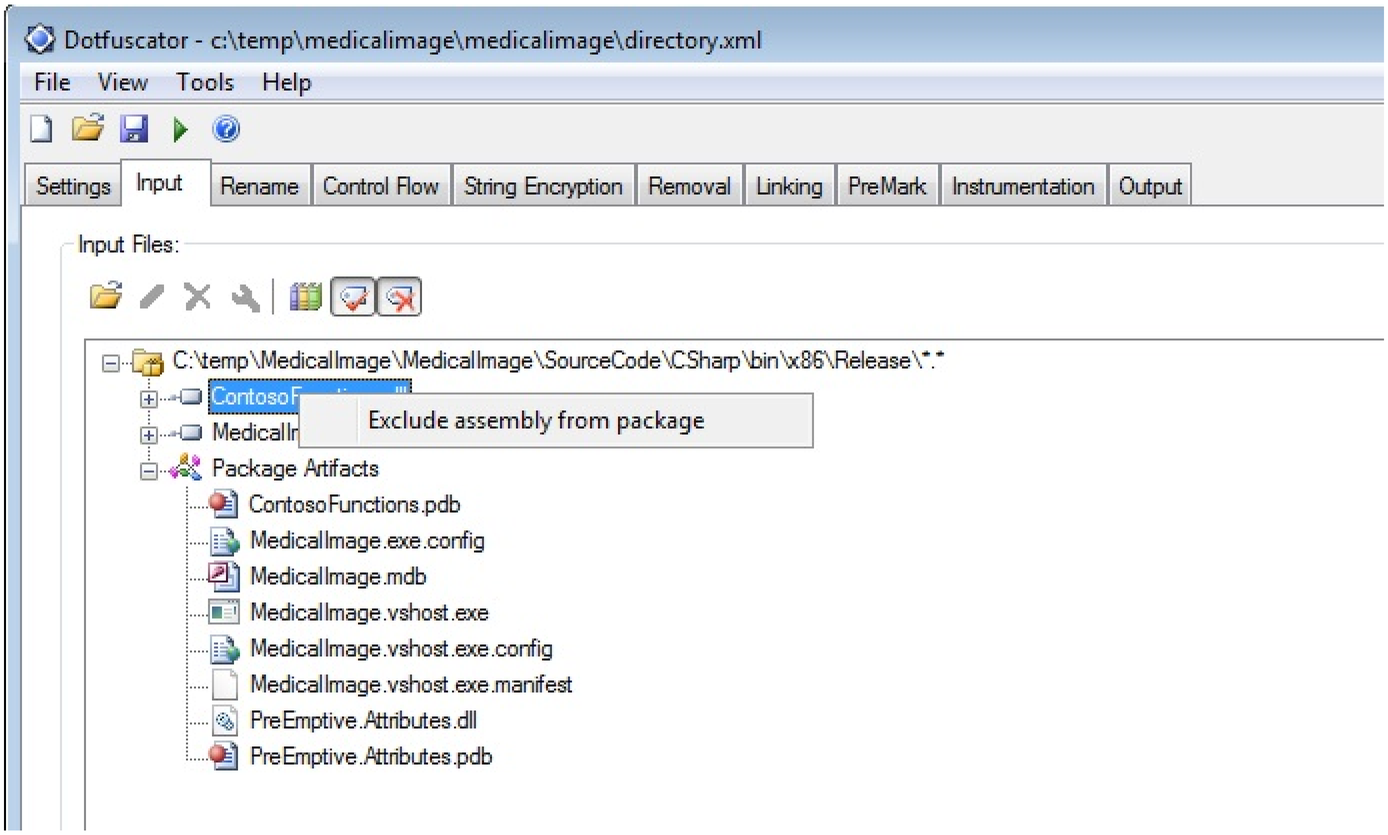
Transform XAML Mode

Transform XAML mode can be toggled for all assemblies using the Transform XAML button on the toolbar.

Alternatively, you can set the XAML Transform mode for specific assemblies by checking or un-checking the **Transform XAML** checkbox under the input assembly's entry in the list.

Excluding or including package assemblies from processing

Specified assemblies contained in packages can be excluded from being round tripped through Dotfuscator by right clicking on the assembly node in the package and selecting **Exclude assembly from package**. This will cause the assembly to be added to the list of package artifacts that are not processed by Dotfuscator. By being added to the artifacts list assemblies are exempt from any obfuscation and instrumentation and all existing strong naming and signing is preserved. This can be switched back by right clicking again and choosing **Include assembly in package**.


Dotfuscator - c:\temp\medicalimage\medicalimage\directory.xml
File View Tools Help
Settings Input Rename Control Flow String Encryption Removal Linking PreMark Instrumentation Output
Input Files:
Exclude assembly from package

```
C:\temp\MedicalImage\MedicalImage\SourceCode\CSharp\bin\x86\Release\*.*
  ContosoF...
  MedicalIm...
  Package Artifacts
    ContosoFunctions.pdb
    MedicalImage.exe.config
    MedicalImage.mdb
    MedicalImage.vshost.exe
    MedicalImage.vshost.exe.config
    MedicalImage.vshost.exe.manifest
    PreEmptive.Attributes.dll
    PreEmptive.Attributes.pdb
```

Declarative Obfuscation

The **Honor Obfuscation Attributes** and **Strip Obfuscation Attributes** settings can be toggled for all input assemblies using the respective button on the toolbar.

Alternatively, you can configure these settings for specific assemblies by checking or un-checking the appropriate checkbox under the input assembly's entry in the list.

Instrumentation Attributes

The **Honor Instrumentation Attributes** and **Strip Instrumentation Attributes** settings can be configured for specific assemblies by checking or un-checking the appropriate checkbox under the input assembly's entry in the list. See Configuring and Running Dotfuscator with Application Analytics for details about these settings.

2.5.1.2.1 Directory Inputs

Dotfuscator provides the ability to obfuscate and/or instrument all files in a directory via a Directory Package input. A Directory Package consists of a relative or absolute path to a directory and optionally a wildcard specifier (file mask) of which files to match. All managed assemblies that match the file mask will be used as inputs to Dotfuscator. Any unmanaged assemblies or other files that match the file mask will be listed as Package Artifacts and, while not processed by Dotfuscator, will be copied to the output directory during the build process.

To add a Directory Package select *Add Input* and type the path and a file mask wildcard to the *Add Input* dialog box. You can also add an entire directory of files (*.*) by selecting the *Browse* button, navigating to the directory of your choice and leaving text *Folder Select* in the file name prompt. You can specify an explicit path or use a Project Property to specify a substitution property for all or part of the path.

All project settings will be applied to all Directory Package assembly inputs and exclusion rules can be created and saved in the Dotfuscator project for any assemblies contained in the list of package assemblies.

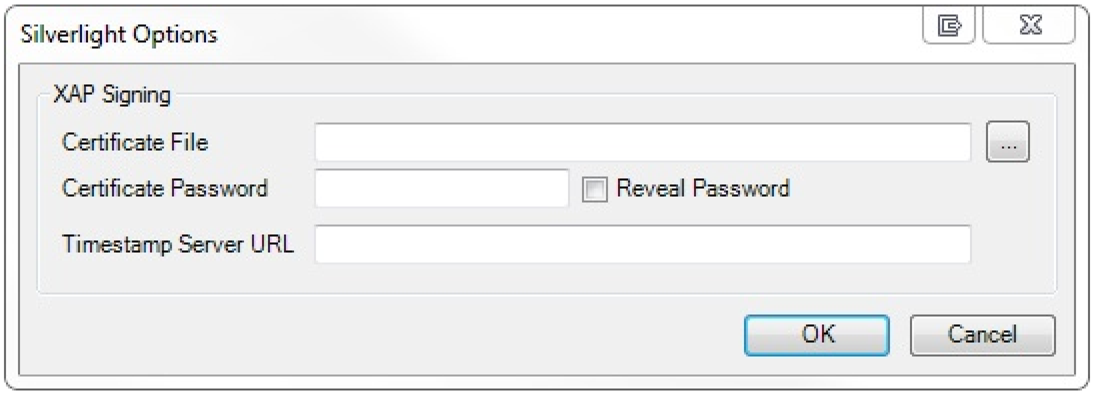
2.5.1.2.2 Silverlight Inputs

Dotfuscator provides the ability to specify a Silverlight deployment file, *.XAP*, as an input. A Silverlight Package consists of a single XAP file which will be parsed and presented in the user interface as a Silverlight Package. All managed assemblies that are contained in the XAP will be used as inputs to Dotfuscator. Any other files contained in the XAP will be listed as Package Artifacts and, while not processed by Dotfuscator, will be included in the output XAP in the output directory. Dotfuscator will output a single XAP that contains obfuscated and/or instrumented assemblies, an updated manifest and any other non-assembly files from the input XAP.

To add a Silverlight Package select Add Input and type the path and a file name in the *Add Input* dialog box. You can also browse to the specific XAP file by selecting the *Browse* button and navigating to it. You can specify an explicit path or use a Project Property to specify a substitution property for all or part of the path.

All project settings will be applied to all Silverlight Package assembly inputs and exclusion rules can be created and saved in the Dotfuscator project for any assemblies contained in the list of package assemblies.

A Silverlight 4 package that is signed can be re-signed by Dotfuscator. The signing options are accessed via the *Package Properties* button or context menu entry and consists of the **Certificate File, Certificate Password**, and **Timestamp Server URL** settings. The Silverlight certificate options set the certificate file container and optional certificate password and timestamp server that are used to sign the output XAP file.



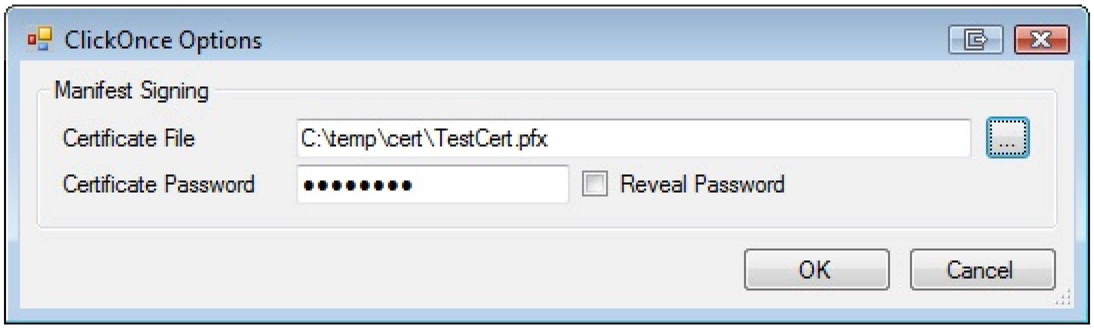
2.5.1.2.3 ClickOnce Inputs

Dotfuscator provides the ability to specify a ClickOnce deployment manifest file, *.APPLICATION*, as input. A ClickOnce Package consists of a single .APPLICATION file which will be parsed and presented in the user interface as a ClickOnce Package. All managed assemblies that are contained in the application will be used as inputs to Dotfuscator. Any other files contained in the deployment and application manifests will be listed as Package Artifacts and, while not processed by Dotfuscator, will be included in the output application in the output directory. Dotfuscator will output to the output directory all obfuscated and/or instrumented assemblies, all necessary manifests, and any other non-assembly files from the input manifests.

To add a ClickOnce Package select *Add Input* and type the path and a file name in the *Add Input* dialog box. You can also browse to the specific deployment manifest file by selecting the *Browse* button and navigating to it. You can specify an explicit path or use a Project Property to specify a substitution property for all or part of the path.

All project settings will be applied to all ClickOnce Package assembly inputs and exclusion rules can be created and saved in the Dotfuscator project for any assemblies contained in the list of package assemblies.

A ClickOnce Package has two required properties that are accessed via the *Package Properties* button or context menu entry, the **Certificate File** and **Certificate Password** settings. The ClickOnce certificate options refer the certificate and optional certificate password that are used to sign all of the output manifest files.


ClickOnce Options

Manifest Signing

Certificate File: C:\temp\cert\TestCert.pfx

Certificate Password: ••••••••

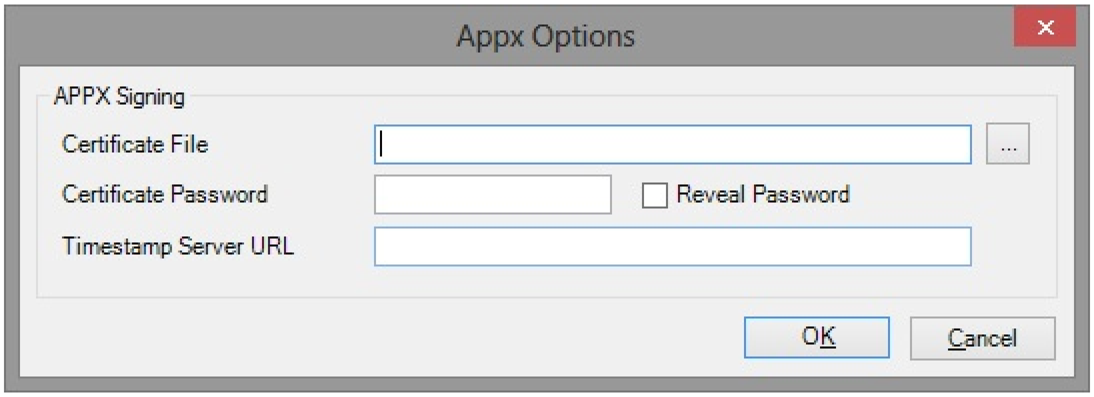
2.5.1.2.4 Windows Store Inputs

Dotfuscator provides the ability to specify a Windows Store deployment file, *.APPX*, as an input. A Windows Store Package consists of a single APPX file which will be parsed and presented in the user interface as a Windows Store Package. All managed assemblies that are contained in the APPX will be used as inputs to Dotfuscator. Any other files contained in the APPX will be listed as Package Artifacts and will be included in the output APPX in the output directory. Dotfuscator will output a single APPX that contains obfuscated and/or instrumented assemblies, an updated manifest and any other non-assembly files from the input APPX.

To add a Windows Store Package select Add Input and type the path and a file name in the *Add Input* dialog box. You can also browse to the specific APPX file by selecting the *Browse* button and navigating to it. You can specify an explicit path or use a Project Property to specify a substitution property for all or part of the path.

All project settings will be applied to all Windows Store Package assembly inputs and exclusion rules can be created and saved in the Dotfuscator project for any assemblies contained in the list of package assemblies.

All Windows Store packages are signed and must be re-signed by Dotfuscator. The signing options are accessed via the *Package Properties* button or context menu entry and consists of the **Certificate File, Certificate Password**, and **Timestamp Server URL** settings. The APPX certificate options set the certificate file container and optional certificate password and timestamp server that are used to sign the output APPX file.


Appx Options

APPX Signing

Certificate File

Certificate Password ☐ Reveal Password

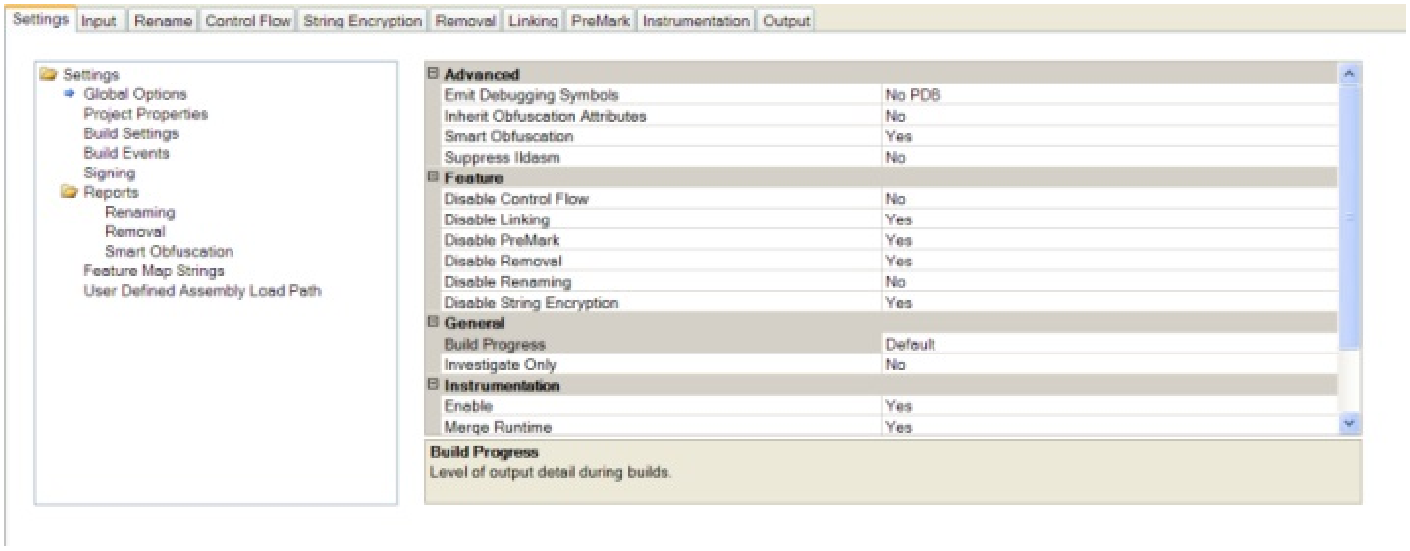
Timestamp Server URL

2.5.1.3 The Settings Tab

The *Settings* tab allows you to configure *global options*, *project properties*, *build settings* and *events*, code *signing*, *reporting*, *feature map strings* for declarative obfuscation, and user-defined *assembly load paths*. You can choose the feature you wish to edit by selecting the appropriate node in the navigation pane on the left side of the tab.

Global Options

The global options editor allows you to set the global options for the project.



- Settings
  - Global Options
  - Project Properties
  - Build Settings
  - Build Events
  - Signing
  - Reports
    - Renaming
    - Removal
    - Smart Obfuscation
  - Feature Map Strings
  - User Defined Assembly Load Path

Advanced	
Emit Debugging Symbols	No PDB
Inherit Obfuscation Attributes	No
Smart Obfuscation	Yes
Suppress Ildasm	No
Feature	
Disable Control Flow	No
Disable Linking	Yes
Disable PreMark	Yes
Disable Removal	Yes
Disable Renaming	No
Disable String Encryption	Yes
General	
Build Progress	Default
Investigate Only	No
Instrumentation	
Enable	Yes
Merge Runtime	Yes

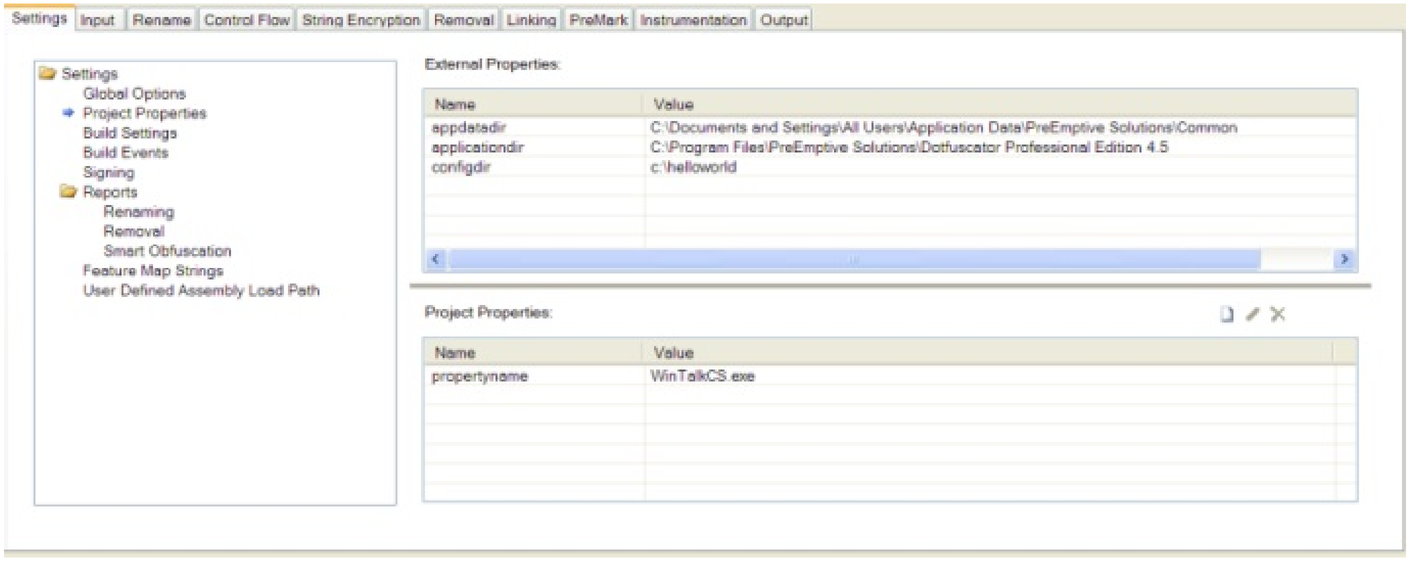
Build Progress
Level of output detail during builds.

You can selectively enable or disable Dotfuscator's features, such as renaming, from this tab. Additionally, you may also modify the following options:

- **Emit Debugging Symbols.** Emit debugging symbols for obfuscated assemblies and control JIT behavior. See Debug Global Option.
- **Inherit Obfuscation Attributes.** This option specifies whether declarative obfuscation attributes applied to types and methods should also be applied to derived types and overriding methods.
- **Smart Obfuscation.** This allows you to enable or disable automated renaming and removal exclusions for selected application types. See Smart Obfuscation for more details. By default, it is enabled.
- **Suppress Ildasm**. This tells Dotfuscator to add the SuppressIldasmAttribute to all output assemblies which will prevent Microsoft's Ildasm utility from displaying the assembly's IL. This is only valid for assemblies targeting .NET 2.0 and above.
- **Disable [feature].** Dotfuscator allows you to enable or disable each of its transforms. The transforms that are Disabled by default in new projects include: Linking, Watermarking, Removal, and String Encryption. Transforms that are Enabled by default in new projects include Renaming, Control Flow and Instrumentation.
- **Build Progress**. This controls the verbosity of Dotfuscator's output during a build.
- **Investigate Only**. This tells Dotfuscator to generate reports but no output assemblies.
- **Enable Instrumentation**. This allows you to enable or disable the ability to instrument your application which will process and remove instrumentation attributes. By default, instrumentation is enabled.
- **Merge Runtime**. This allows you to embed the PreEmptive Analytics library in one of the output assemblies rather than referencing it as a separate DLL. By default, this behavior is enabled.
- **Send Analytics Messages**. This allows you to instrument the application so that it will send application lifecycle, session lifecycle, and feature use messages to a PreEmptive Analytics Endpoint. See Configuring and Running Dotfuscator with Application Analytics.
- **Send Shelf Life Messages**. This allows you to instrument the application so that it will send shelf life warning, shelf life expiration, and sign of life messages to a PreEmptive Analytics Endpoint. See Configuring and Running Dotfuscator with Shelf Life.
- **Send Tamper Messages**. This allows you to instrument the application so that it will send tamper notification messages to a PreEmptive Analytics Endpoint. See Configuring and Running Dotfuscator with Application Analytics.

Project Properties

The properties editor allows you to view and add user-defined name-value pairs as *Project Properties* and to view *External Properties* that have been defined from the command line. See Property List and Properties for a full explanation. To add a Project Property, click the **New** button on the project properties toolbar.



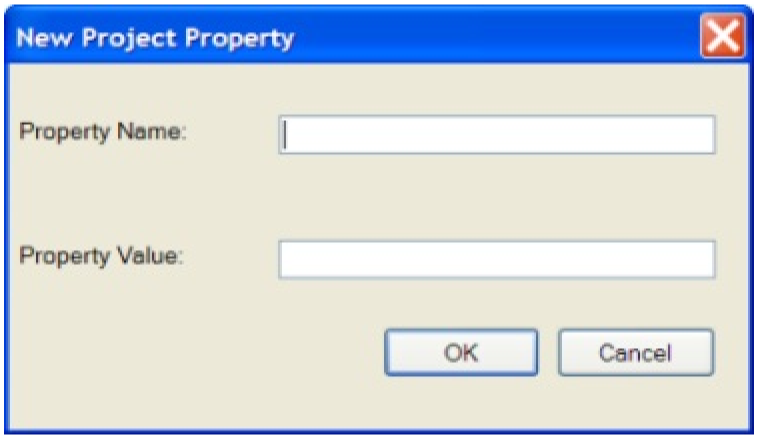
External Properties:

Name	Value
appdatadir	C:\Documents and Settings\All Users\Application Data\PreEmptive Solutions\Common
applicationdir	C:\Program Files\PreEmptive Solutions\Dotfuscator Professional Edition 4.5
configdir	c:\helloworld

Project Properties:

Name	Value
propertyname	WinTalkCS.exe

This brings up the *New Project Property* dialog. Type in a **name** and a **value** in the fields provided. Click on the **OK** button, and the property will be set.



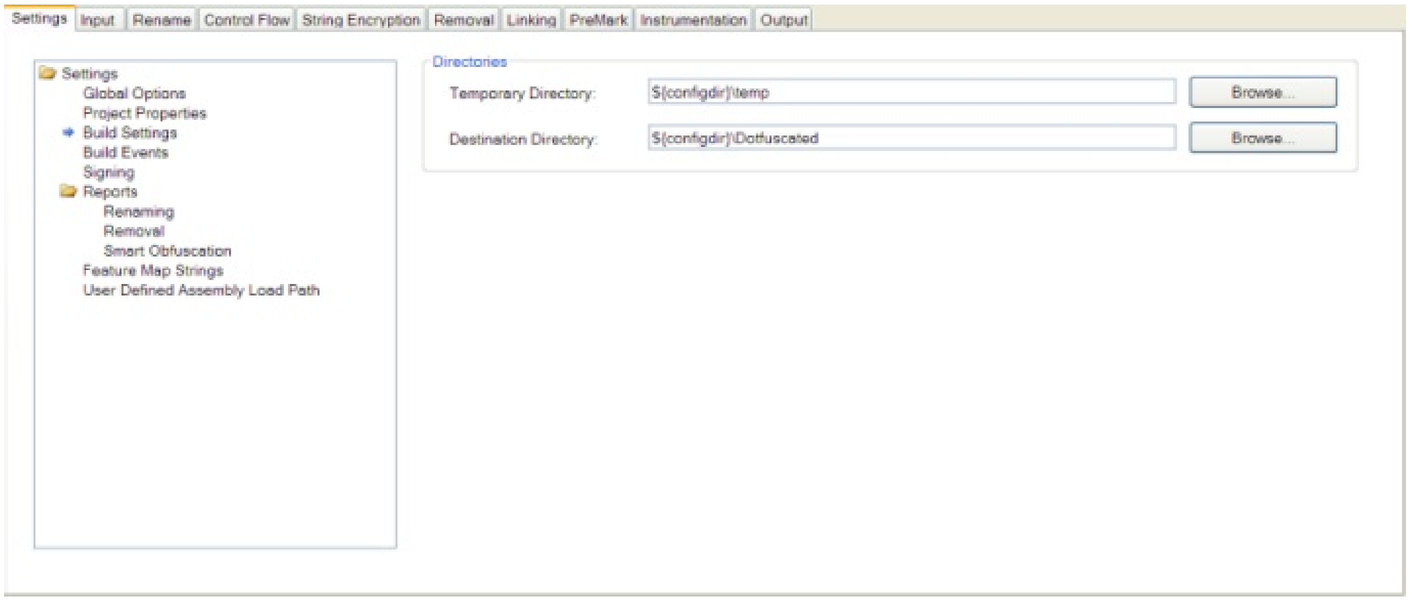
You can use the **Edit** and **Delete** toolbar buttons in a similar manner to modify or remove Project Properties. You can also delete a property by selecting it and pressing the **Delete** key.

Build Settings

The build settings editor is where the project's destination directory, and optional temporary directory, are established.

When you create a new project, the output directory is set by default to **${configdir}\Dotfuscated**. The **${configdir}** is a built-in project property that is expanded to the folder that contains your project file.

If you want to choose a different destination directory or establish a temporary directory, enter the path to the directory in the appropriate text field.



Temporary Directory: ${configdir}\temp

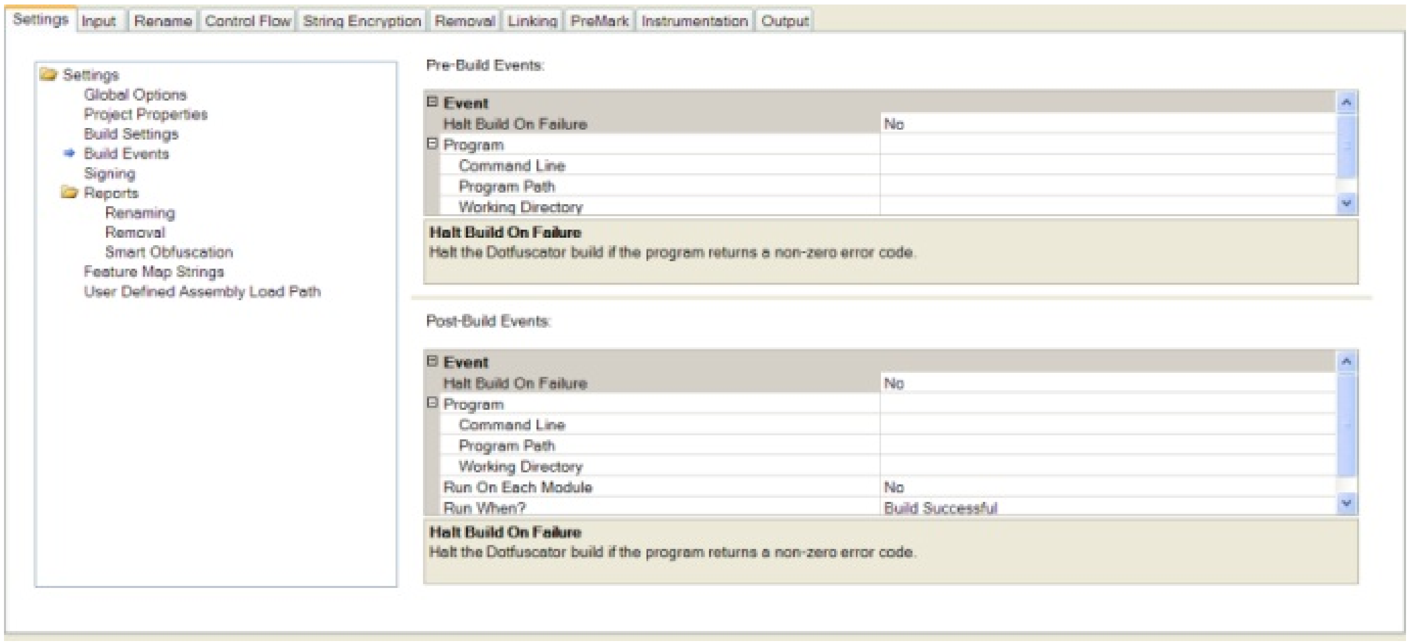
Destination Directory: ${configdir}\Dotfuscated

The *Temporary Directory* is optional and is used to store temporary files during processing. By default Dotfuscator will use your Windows temporary directory. If you wish to specify this directory, enter the **path** to the directory in this field or click **Browse** to choose its location graphically.

The *Destination Directory* is required and specifies where the output from the build will reside. Enter the **path** to the directory in this field or click **Browse** to choose its location graphically.

Build Events

The *Build Events* property page is where you specify Build Events for your Dotfuscator project.



Pre-Build Events:

Event	
Halt Build On Failure	No
Program	
Command Line	
Program Path	
Working Directory	

Halt Build On Failure
Halt the Dotfuscator build if the program returns a non-zero error code.

Post-Build Events:

Event	
Halt Build On Failure	No
Program	
Command Line	
Program Path	
Working Directory	
Run On Each Module	No
Run When?	Build Successful

Halt Build On Failure
Halt the Dotfuscator build if the program returns a non-zero error code.

For each event you can specify an external program that runs when the event occurs. You can also specify a working directory and command line options for the program, and whether Dotfuscator should halt the build (fail) if the specified program returns a non-zero error code.

For the post-build event, you can specify under what conditions it will run (e.g. all the time, only if the build succeeds, or only if the build fails). You can also specify whether you want the post build event to run only once for the project, or run once for each output module.

Signing

Strong Naming

The signing editor allows you to configure Dotfuscator to automatically sign or resign your strongly named assemblies. See Dotfuscating Strong Named Assemblies for more information.

Strong named assemblies are digitally signed. This allows the runtime to determine if an assembly has been altered after signing. The signature is an SHA1 hash signed with the private key of an RSA public/private key pair. Both the signature and the public key are embedded in the assembly’s metadata.

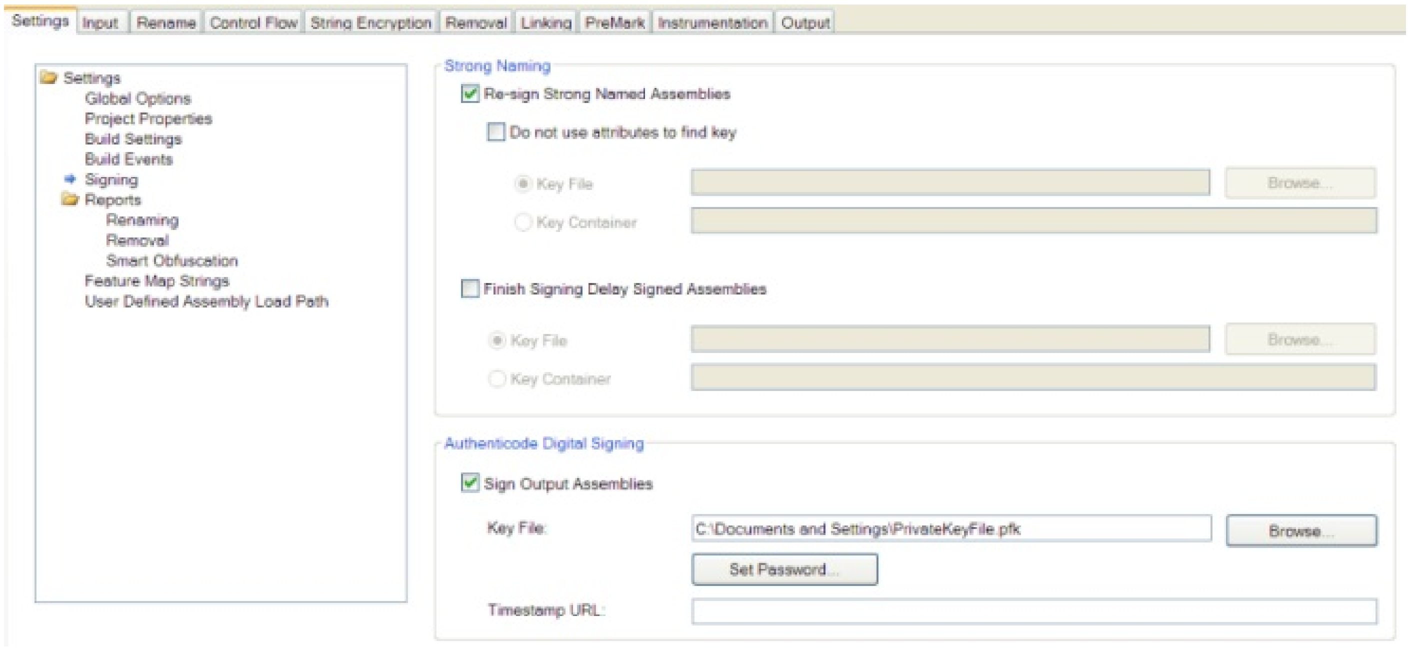
Since Dotfuscator modifies the assembly, it is essential that signing occur *after* running the assembly through Dotfuscator.

Dotfuscator handles this step as part of the obfuscation process.

Authenticode Digital Signing

The Authenticode Digital Signing option allows you to attach an Authenticode digital signature to your application. Similar to a security certificate, this signature certifies that the application you are obfuscating and instrumenting is your intellectual property, and allows users to ensure that the resulting binaries were provided by you alone and have not been modified. This feature adds another level of security to safeguard your application. To attach an Authenticode signature to your output assemblies, check the *Sign Output Assemblies* checkbox and then click the *Browse...* button to locate your **Key File**, or enter its path in the text box. Once the *Key File* field is properly populated, click the *Set Password...* button to set the password for your Key File.

The Timestamp URL field provides the ability for you to specify the URL of an Authenticode timestamp service. This URL will be accessed during Dotfuscator's signing process, and will provide additional data which will allow your assemblies' Authenticode signatures to remain valid after your code-signing certificate has expired. This element is optional. If omitted, this additional data will not be included, and your assemblies' Authenticode signatures will become invalid once your code-signing certificate expires.



[✓] Re-sign Strong Named Assemblies

[ ] Do not use attributes to find key

[ ] Finish Signing Delay Signed Assemblies

[✓] Sign Output Assemblies

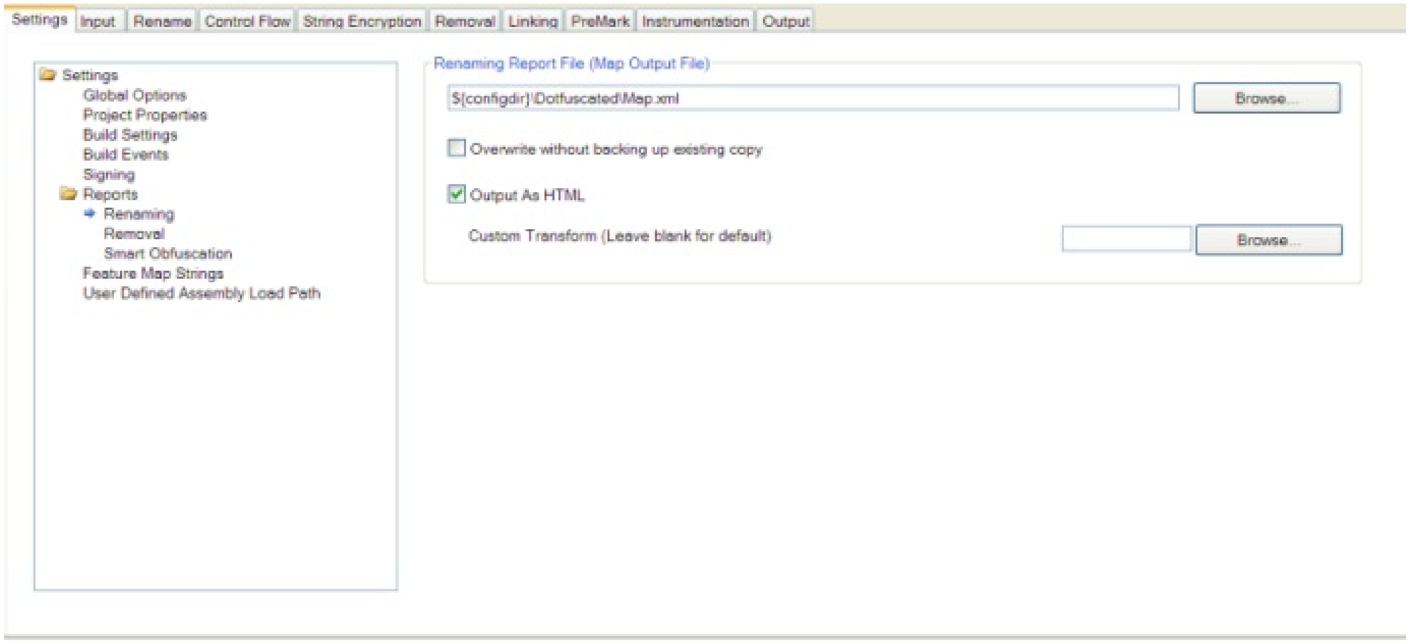
Key File: C:\Documents and Settings\PrivateKeyFile.pfk

Renaming Reports

The renaming report provides a summary of all the elements renamed by Dotfuscator during a specific run, including a statistics section. The *Renaming Report File* (*Map Output File)* section allows you to specify the location to save a renaming map file. You may leave the default value, or enter your preferred path. If you know the name and path of the mapping file you want to use, you can type it directly into the text box. Alternatively, you can browse your file system for the intended file location. The **Browse** button on the right of the text box brings up the *Select Map Output File* window that provides a familiar navigational dialog.

You also have the option of overwriting the output file each time you build the application without generating a backup of the existing copy of the output.

Dotfuscator has a default transform that can generate a readable HTML formatted version of the report in addition to the default .XML format. If you do check the *Output As HTML* box, the *Custom Transform (Leave blank for default)* field is activated. As the field name states, leave this blank for the default custom transform or click **Browse** to select the location of your choice.



Renaming Report File (Map Output File)

${configdir}\Dotfuscated\Map.xml

☐ Overwrite without backing up existing copy

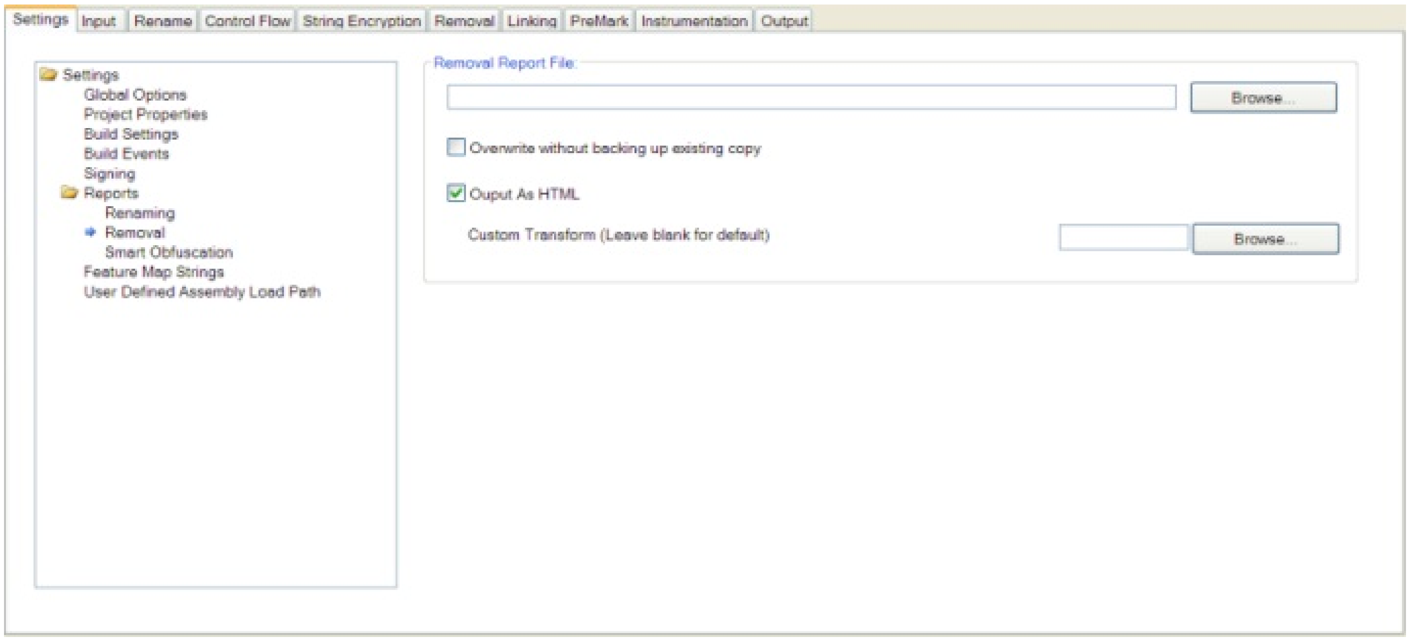
☑ Output As HTML

Custom Transform (Leave blank for default)

Removal Report

The removal report provides a summary of all the elements removed by Dotfuscator during a specific run, including a statistics section. The *Removal Report File* section allows you to specify the location to save the report. Two approaches are available. If you know the name and path of the file you want to use, you can type it directly into the text box. Alternatively, you can browse your file system for the intended file location. The **"..."** (ellipsis) button on the right of the text box brings up the *Select Removal Report File* window that provides a familiar navigational dialog.

You also have the option of overwriting the output file each time you build the application without generating a backup of the existing copy of the output. The removal report can also be written out as a readable, HTML formatted document, in addition to the XML formatted version. This report provides a quick cross reference that allows you to quickly navigate through all types, fields, and methods to see at a glance which were removed. Checking the *Output as HTML* checkbox tells Dotfuscator to write this report using the same name and path information as the XML version. If the default HTML report doesn't meet your reporting requirements, you can provide your own XSL document that Dotfuscator uses to transform the XML version. If you do check the *Output As HTML* box, the *Custom Transform (Leave blank for default)* field is activated. As the field name states, leave this blank for the default custom transform or click **Browse** to select the location of your choice.



Overwrite without backing up existing copy [ ]

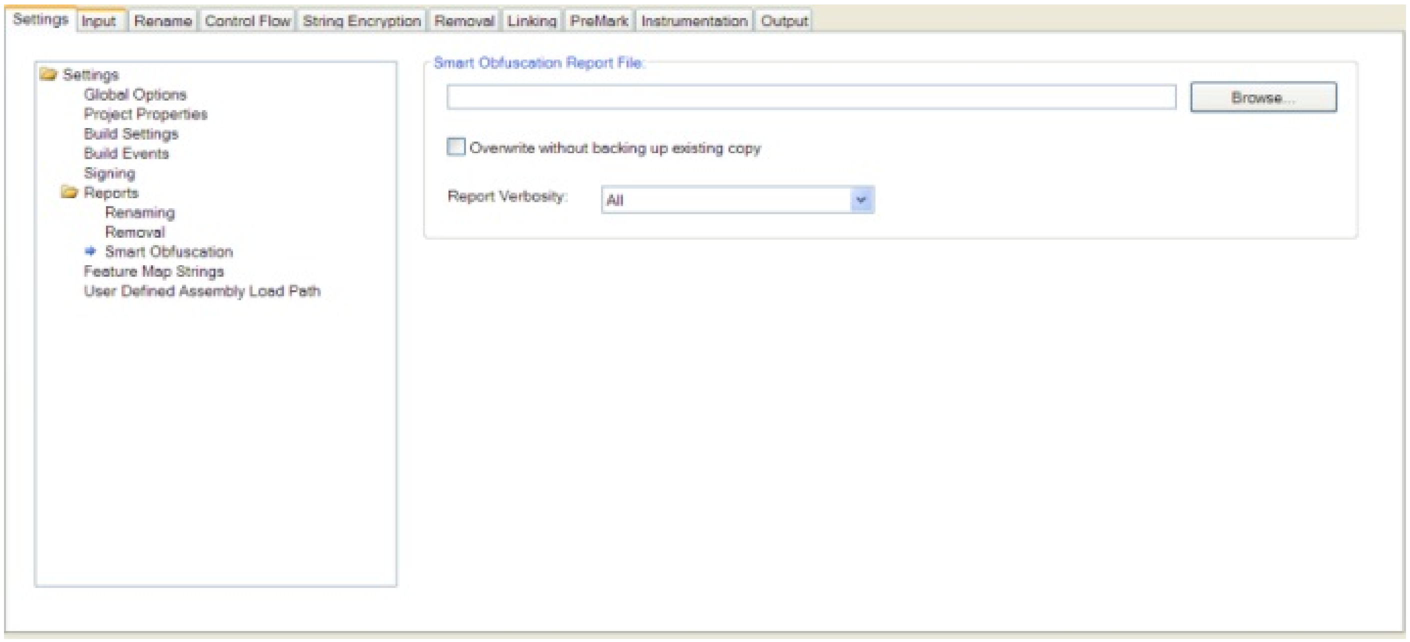
Ouput As HTML [✓]

Custom Transform (Leave blank for default)

Smart Obfuscation Report

The smart obfuscation report provides a summary of all the elements that could not be renamed or removed based on rules provided by the smart obfuscation feature. If items are excluded by smart obfuscation but no destination for the report is specified, the report will appear in a *Smart Obfuscation Report* tab next to the Build Output tab at the bottom of Dotfuscator's window. The *Smart Obfuscation Report File* field allows you to specify the location to save the report. Two approaches are available. If you know the name and path of the mapping file you want to use, you can type it directly into the text box. Alternatively, you can browse your file system for the intended file location. The **Browse** button on the right of the text box brings up the *Select Smart Obfuscation Report File* window that provides a familiar navigational dialog.

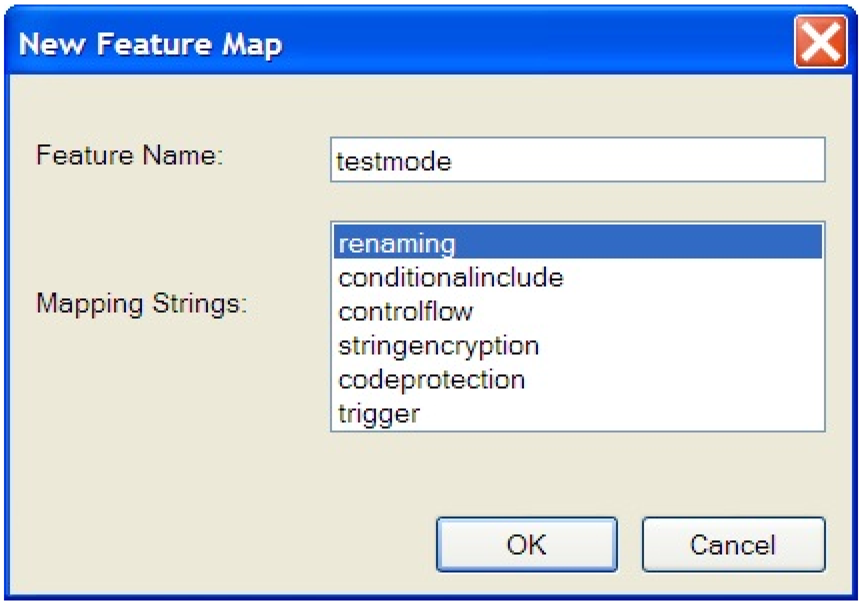
You have the option of overwriting the output file each time you build the application without generating a backup of the existing copy of the output. You may also optionally configure the verbosity of the report. Allowed values for the verbosity attribute are **All**, **Warnings Only**, and **None**. The default value is **All**.



Feature Map Strings

The feature map editor is for Declarative Obfuscation. Declarative Obfuscation is implemented via attribute decoration within the source code while it is being written. The attributes that are used to control Declarative Obfuscation are **System.Reflection.ObfuscateAssemblyAttribute** and **System.Reflection.ObufscationAttribute**. System.Reflection.ObfuscateAssemblyAttribute controls the obfuscation of the assembly as a whole. System.Reflection.ObufscationAttribute controls the obfuscation of individual types and their members. Feature Map Strings enable you to declare, within the source code, what should and should not be obfuscated by using attributes. For a complete description of Dotfuscator's support for Declarative Obfuscation, see Declarative Obfuscation via Custom Attributes. That section describes the *Feature Map* and lists the native feature strings that Dotfuscator understands.

From the toolbar, you can add, edit, and remove feature map strings. The **Add** button brings up a dialog that allows you to map feature strings to supported Dotfuscator features.



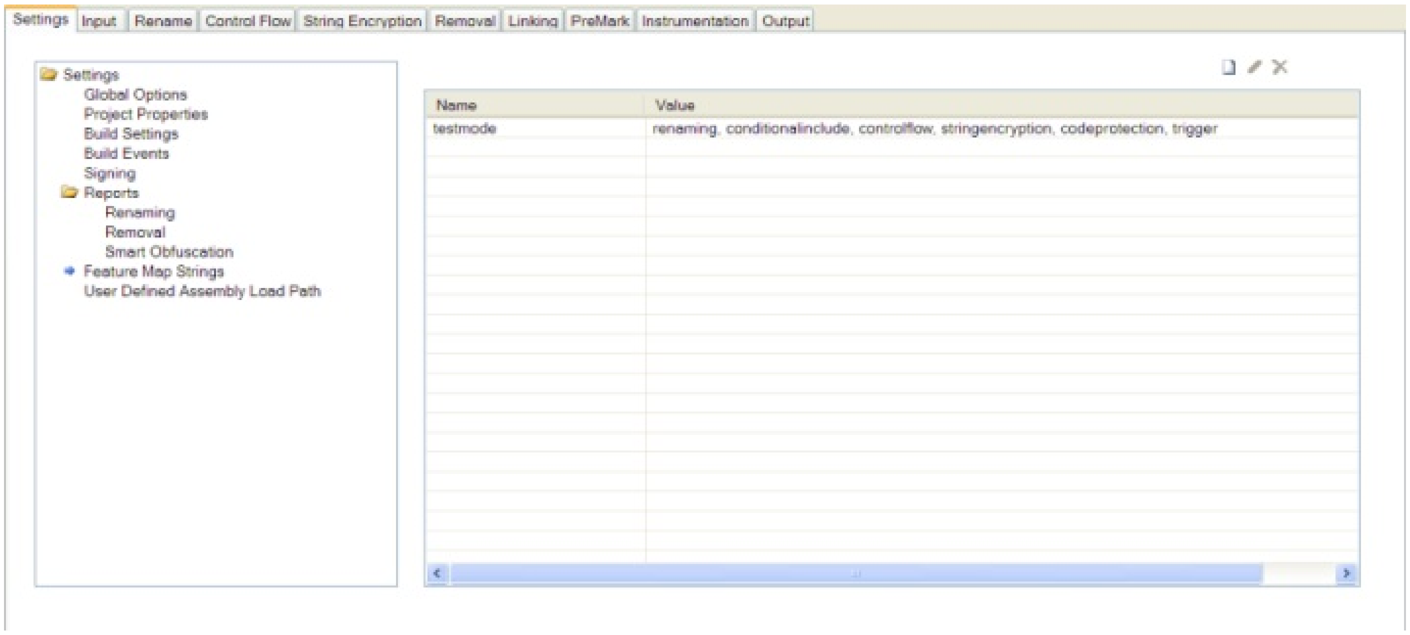
Feature Name: testmode

Mapping Strings: renaming, conditionalinclude, controlflow, stringencryption, codeprotection, trigger

The name that entered in the *Feature Name*: field is the name of the string. The selected Mapping Strings denote the configuration the attribute decorated with that feature will produce:

- **renaming** - attribute configures renaming obfuscation.
- **controlflow** - attribute configures control flow obfuscation.
- **stringencryption** - attributes configures stringencryption.
- **trigger** - attribute configures pruning by marking the annotated item as an entry point.
- **conditionalinclude** - attribute configures pruning by conditionally including the annotated item.

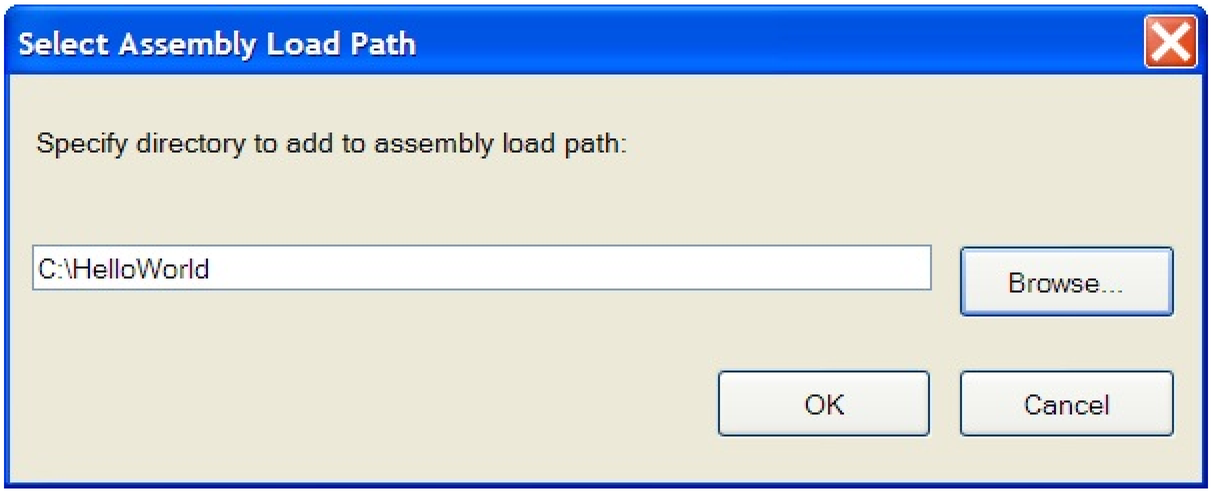
When a feature is selected, the **Edit** and **Delete** buttons are activated. Selecting **Edit** brings up a dialog that allows you to edit the map feature strings to supported Dotfuscator features. You can also delete a feature map string by selecting it and pressing the **Delete** key.



Name	Value
testmode	renaming, conditionalinclude, controlflow, stringencryption, codeprotection, trigger

User Defined Assembly Load Path

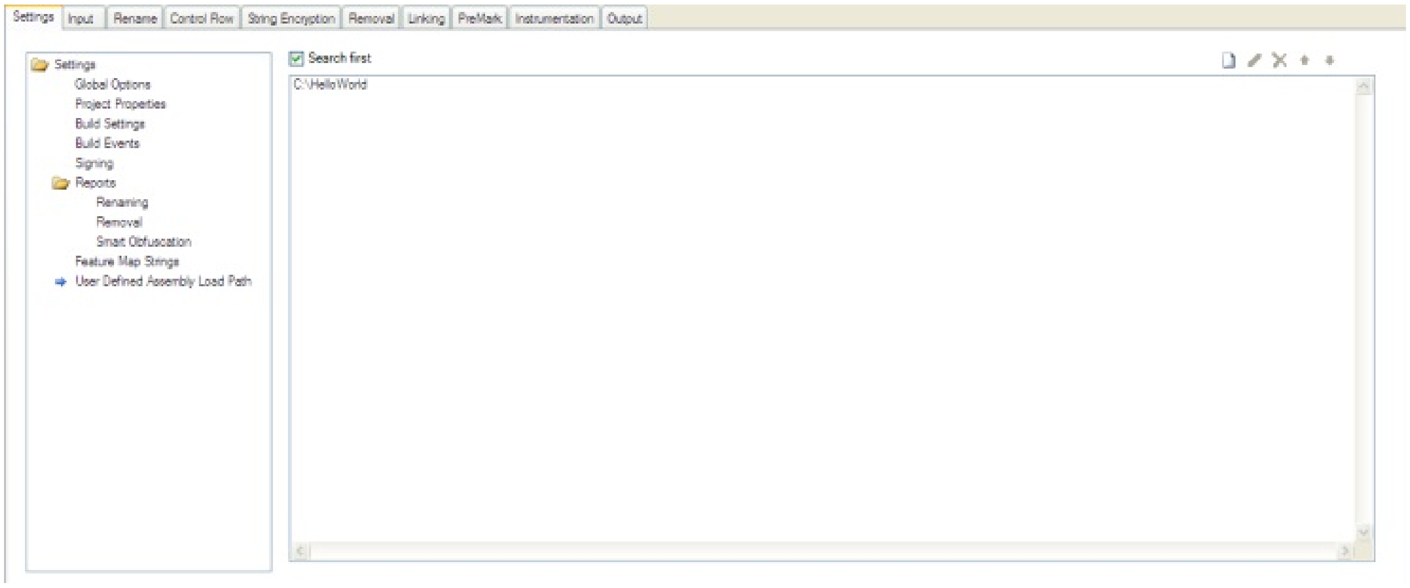
Using this editor, you can edit your project's user defined assembly load paths. From the toolbar, you can add or remove directories, edit existing directories, or change the order in which they are searched. When you click **Add**, this window displays:



Specify directory to add to assembly load path:

C:\HelloWorld

Two approaches are available for specifying the assembly load path. If you know the name and path you want to use, you can type it directly into the text box. Alternatively, you can browse your file system for the intended file location. The **Browse** button on the right of the text box brings up the *Browse for Folder* window that provides a familiar navigational dialog. When the path is specified, click **OK**. The path now displays in the User Defined Assembly Load Path window:



- [✓] Search first

```
C:\HelloWorld
```

Check the **Search First** checkbox to have Dotfuscator search the load path before applying its standard search. When unchecked, Dotfuscator will search the loadpath only after applying its standard search.

2.5.1.4 Configuring

The standalone user interface allows you to configure settings independently for each feature. Each feature has an editor that has its own tab on the standalone user interface. The editors are fully described in the following topics:

- Renaming Editor
- Control Flow Editor
- String Encryption Editor
- Removal Editor
- Linking Editor
- PreMark Editor
- Instrumentation Editor

2.5.1.5 Building the Project

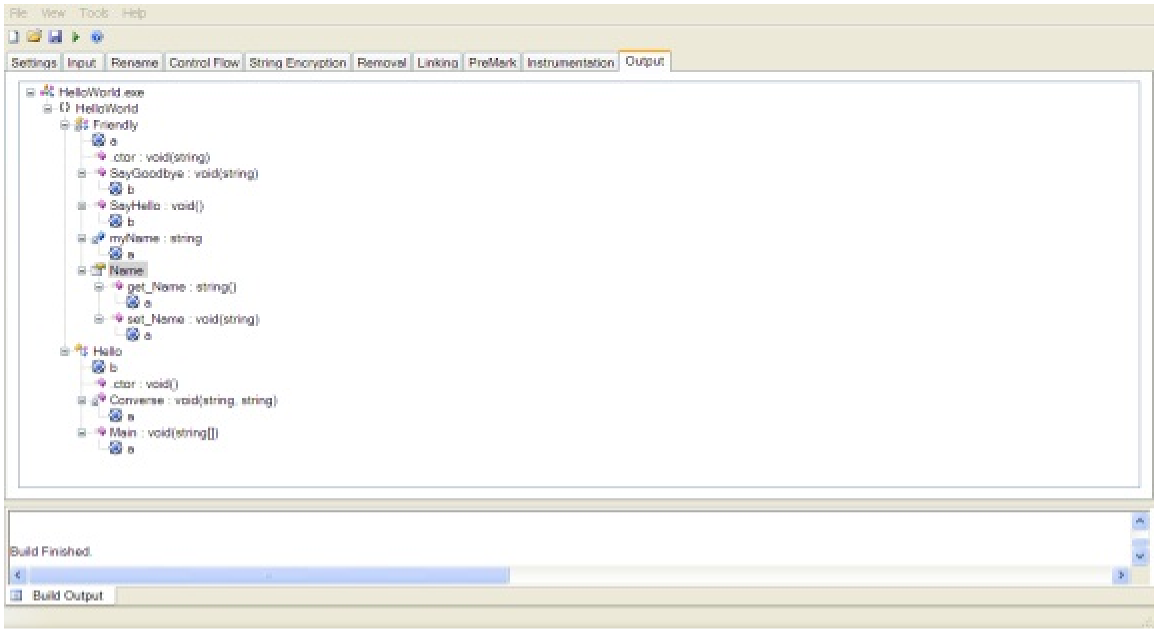
There are two ways to Dotfuscate in the standalone GUI:

- You can click on the **Build Project** button on the Toolbar.
- You can click from the Menu: **File > Build**.

During and after the build, you can see Dotfuscator’s output in the console area.

2.5.1.5.1 The Output Tab

Once your project is Dotfuscated, you can inspect the results from the *Output* Tab.


File View Tools Help
Settings Input Rename Control Flow String Encryption Removal Linking PreMark Instrumentation Output
Build Output

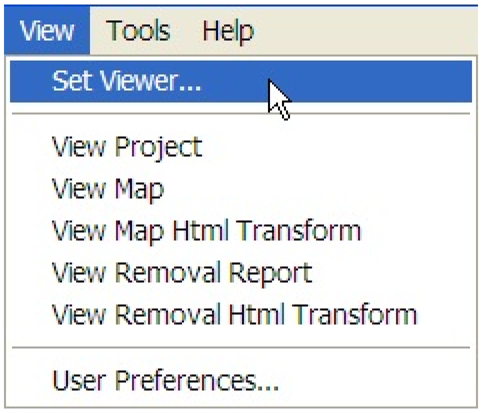
```
HelloWorld.exe
  HelloWorld
    Friendly
      a
      .ctor : void(string)
      SayGoodbye : void(string)
        b
      SayHello : void()
        b
      myName : string
        a
      Name
        get_Name : string()
          a
        set_Name : void(string)
          a
    Hello
      b
      .ctor : void()
      Converse : void(string, string)
        a
      Main : void(string[])
        a
```

```
Build Finished.
```

Here you can browse the tree view and see how Dotfuscator renamed your types, methods, and fields. The new names appear as child nodes under the original nodes in the tree.

2.5.1.6 Viewing Project Files and Reports

Dotfuscator allows you to directly view the XML Configuration, mapping, and report files for your project by loading them in an XML viewer utility of your choice. You specify the utility you want to use by choosing **View > Set Viewer** from the menu. The viewer utility can be any application that displays text files.

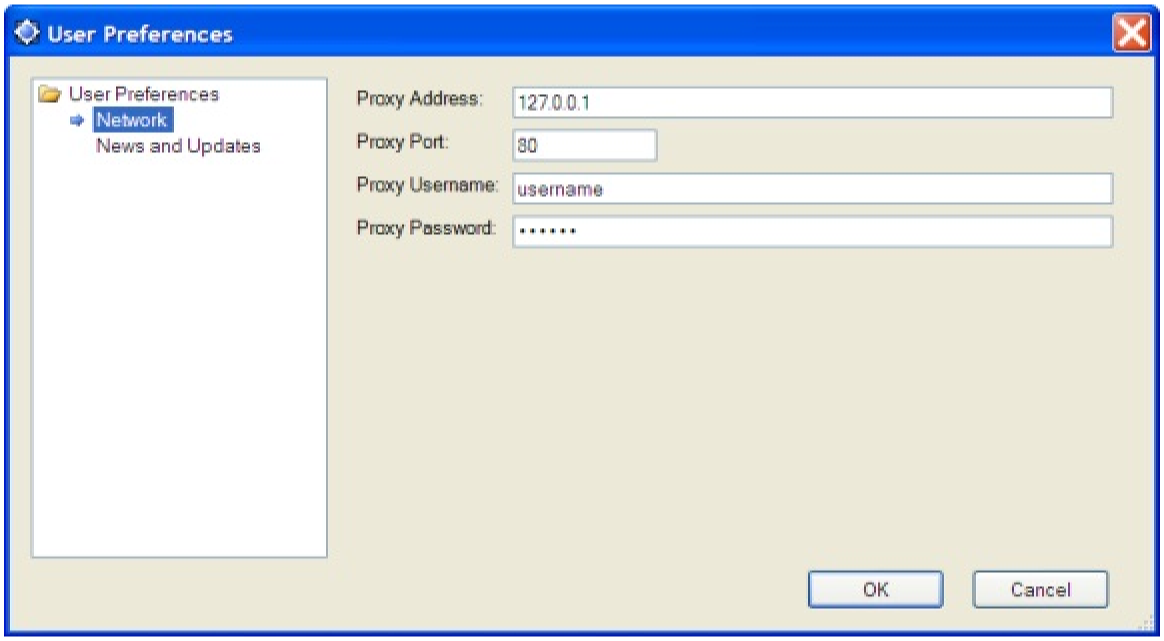


- To load the Project Configuration file in the External Viewer, select **View Project** from the standalone user interface **View** menu.
- To load the Map file in the Viewer, select **View Map** from the standalone user interface **View** menu. This menu option displays the XML version of the report.

- If you opted to generate an HTML version of the map file, you can view it in your default browser by selecting **View Map HTML Transform** from the **View** menu.
- To load the removal report file in the Viewer, select **View Removal Report** from the user interface **View** menu. This menu option will display the XML version of the report.
- If you opted to generate an HTML version of the removal report, you can view it in your default browser by selecting **View Map HTML Transform** from the **View** menu.

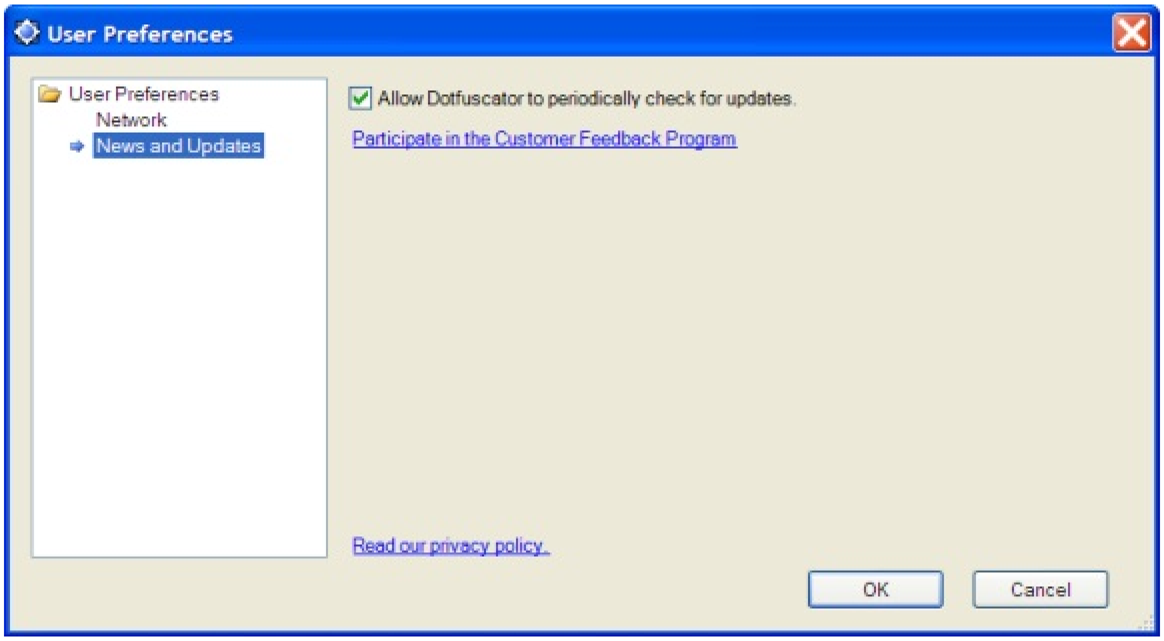
2.5.1.7 Set User Preferences

The **View** menu provides a **User Preferences** option. In the network section of this dialog, the configuration settings of a proxy server for network access may be entered.



Proxy Address: 127.0.0.1
Proxy Port: 80
Proxy Username: username
Proxy Password: ••••••

In the *News and Updates* section of this dialog box, users may opt to allow Dotfuscator to periodically check for updates. A link to our Privacy Policy is included in this dialog.



☑ Allow Dotfuscator to periodically check for updates.

Participate in the Customer Feedback Program

Read our privacy policy.

There is also a link that takes you to the Customer Feedback Program dialog. The text in the link shows your current opt-in/opt-out status. If you are concerned about privacy, click the Read our privacy policy link.

2.5.1.8 The Help Menu

The standalone GUI's help menu contains the following items:

- **Help**. This item brings up Dotfuscator's online help (this document).
- **Register Product**. This item is only enabled if your copy has not yet been registered. When selected, it will bring up Dotfuscator's registration dialog. See Registering and Activating Dotfuscator for more information.
- **Activate Dotfuscator**. This item is only enabled if your Dotfuscator subscription has not been activated, is expired, or is about to expire. See Registering and Activating Dotfuscator for more information.
- **What's New**. When selected, this will launch a browser that will take you to Dotfuscator's home page.
- **Customer Feedback Options**. Dotfuscator provides an anonymous usage reporting system that users can opt-in to.
- **Check For Updates Now**. When selected, Dotfuscator will immediately check the web for updates.
- **About Dotfuscator**. When selected, this will bring up Dotfuscator's *About* box which displays user and version information.

2.5.2 The Visual Studio Interface

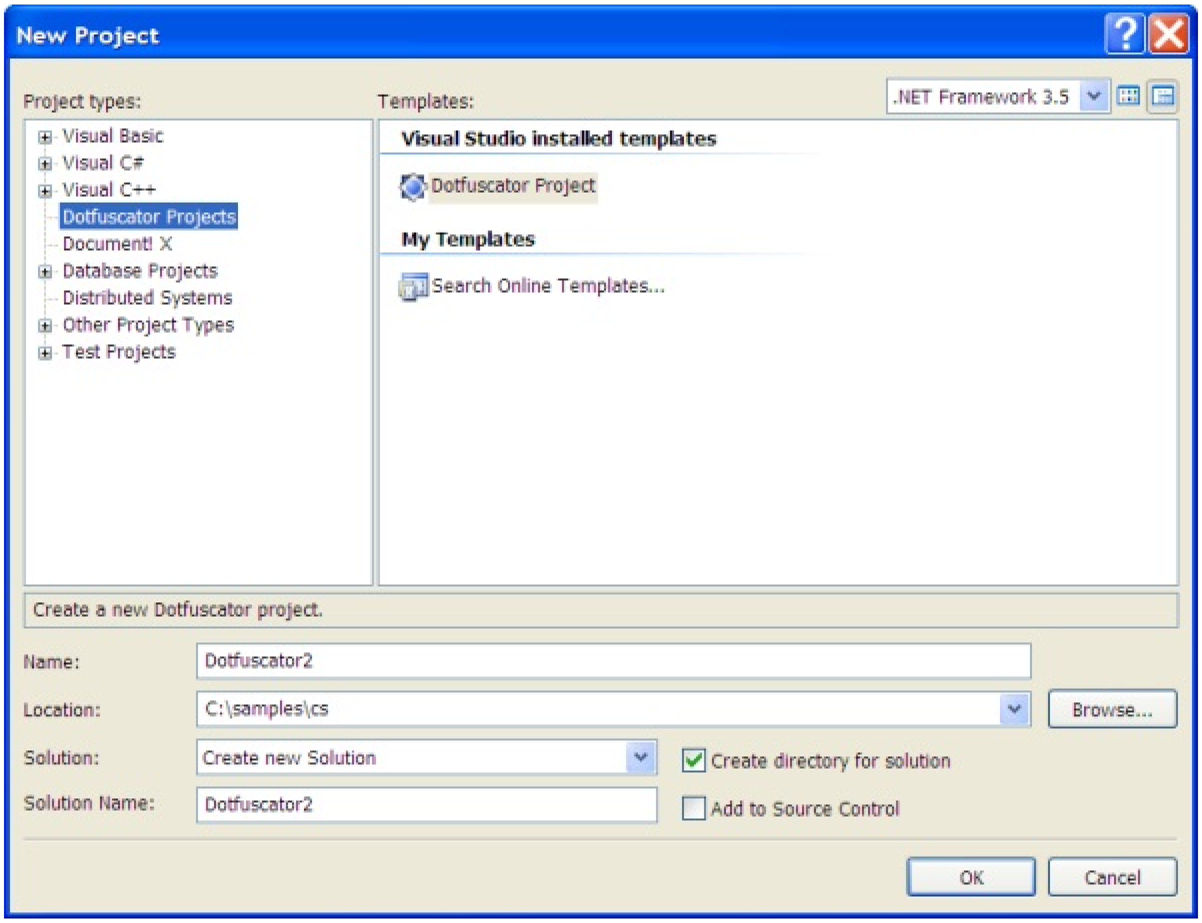
In this section, we describe how to use Dotfuscator when integrated with Visual Studio. Inside Visual Studio, you can create and edit Dotfuscator projects and add them to your existing solutions. During a build, a Dotfuscator project can consume binary outputs (*i.e.* your compiled .NET assemblies) from the development projects in your solution, and in turn, expose the obfuscated outputs to deployment projects.

DMA The Dotfuscator interface within Visual Studio is unavailable in Dotfuscator for Marketplace Apps.

2.5.2.1 Creating a Dotfuscator Project

To create a Dotfuscator project and add it to an existing or new solution, perform the same steps that you would for other Visual Studio project types:

- From the *New Project* dialog, open the **Dotfuscator Projects** folder in the list of *Project Types*.
- Select the **Dotfuscator Project** from the template list.
- Provide a **name** and **location** for your project or accept the defaults.
- In the *Solution:* field, you can choose to create a new solution or add to an existing solution.
- Provide a **name** for your Solution.
- You can select if you want to *Create Directory for solution* and if you want to *Add solution to Source Control* by checking the appropriate box(es).
- Click **OK**.



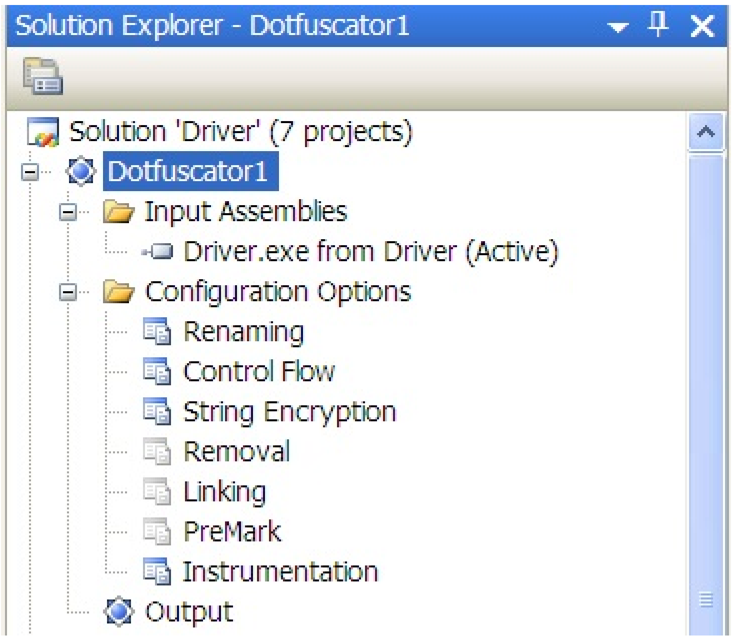
Create a new Dotfuscator project.

Name:	Dotfuscator2
Location:	C:\samples\cs
Solution:	Create new Solution
Solution Name:	Dotfuscator2

[✓] Create directory for solution
[ ] Add to Source Control

2.5.2.2 Solution Explorer and the Dotfuscator Project Tree

When working with Dotfuscator Projects in Visual Studio, *Solution Explorer* is the starting point. Like many other project types, Dotfuscator projects appear as trees in *Solution Explorer*.



The top level project item in the Dotfuscator project tree always has the name that you have given to your Dotfuscator project. You can access the project's property pages from this item through its context menu. See Project Properties for a description of the property pages that a Dotfuscator project provides.

There are always three items immediately under the top level Dotfuscator Project item:

Input Assemblies Folder

This is where you manage the inputs that are to be Dotfuscated. Right clicking on this item gives you a context menu that may be used to add Inputs . Once added, each package and assembly appears as an item in the folder. An input may be removed from the project by selecting the item and pressing the **Delete** key. See Working with Inputs for more information about the different kinds of inputs.

Configuration Options Folder

This folder contains an item for each available configuration editor. You can display the editor by double clicking the appropriate item.

Output Browser Item

After you build your Dotfuscator project, you can browse the output assemblies by double clicking this item. The output browser provides a view of your assemblies similar to that provided by class browser. It shows original and obfuscated symbol names, as well as indicating which symbols were removed from the input assemblies.

2.5.2.3 Project Configurations

With the exception of inputs and their properties, Visual Studio Dotfuscator Project settings are configuration dependent. In other words, you can have different obfuscation settings based on the current active configuration. For example, in a debug build you may want to emit the debugging symbol file, while for a release build you might not.

Dotfuscator project configurations are managed with Visual Studio's Configuration Manager in the same way as other Visual Studio project types. Through the Configuration Manager, you can edit, create, and remove Dotfuscator project configurations. You can also associate them with your solution configurations.

By default, when you create a Dotfuscator project, two project configurations are created for you and associated with the corresponding solution configuration: **Debug** and **Release**. These two configurations are identical at project creation time, except the debug configuration has a project property set to **Emit Debugging Symbols**.

2.5.2.4 Deploying a Dotfuscator Project

Within Visual Studio, a Dotfuscator project exposes its outputs, just like other familiar Visual Studio project types, such as C# or VB.NET. As a result, you can use a Dotfuscator project as a source for any deployment or setup project that knows how to consume Visual Studio projects. This section assumes you are familiar with setup and deployment projects.

Dotfuscator exposes several output groups, summarized below:

Output Group	Description
Primary Output	Contains the output assemblies and any package artifacts
Localized Resource DLLs	Contains satellite DLLs
Reports	Contains removal report and HTML report files
Map Files	Contains XML renaming map file
Debug Symbols	Contains PDB files for output assemblies, if any.

To tell a setup project to package all your obfuscated assemblies, simply point the setup project to the Dotfuscator project’s Primary Output group.

For assemblies that are sourced from other Visual Studio Projects, Dotfuscator also exposes the source project’s dependencies to the deployment project.

2.5.2.5 Working with Inputs

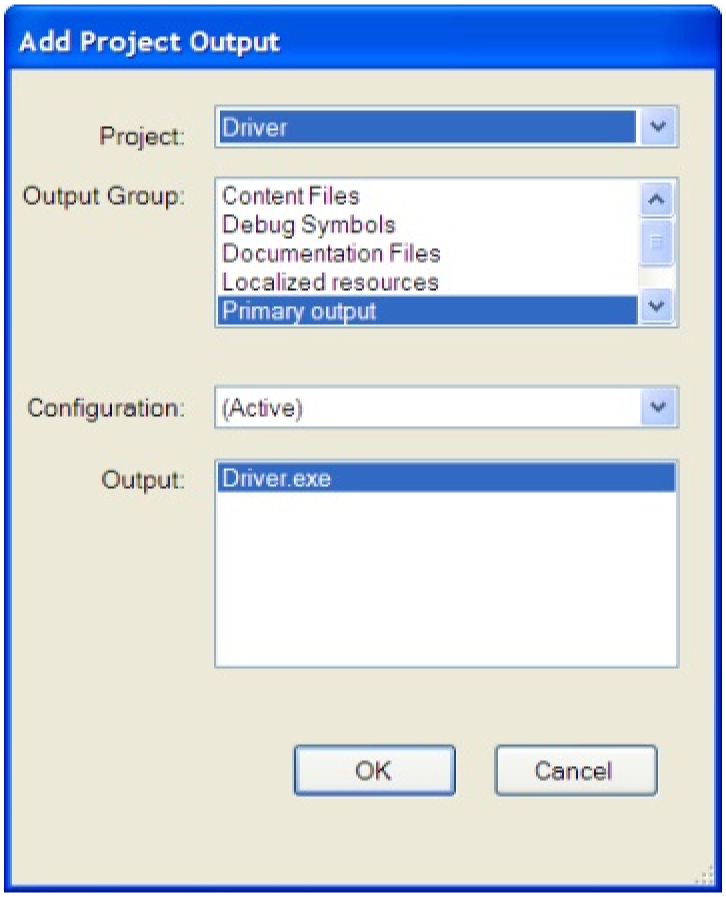
Input packages and assemblies are displayed in the Dotfuscator project tree in Solution Explorer in the *Input Assemblies* folder. You can add inputs by using the context menus associated with the top level project item, the *Input Assemblies* folder item, or from Visual Studio's Project menu.

There are two ways to add input assemblies, depending on whether you want to Dotfuscate an output from another Visual Studio project in your solution, or simply Dotfuscate a package or assembly on your file system.

Input Assemblies from other Projects

To have Dotfuscator use another project's output, you add the input assembly via the **Add Project Output** item that is on the Dotfuscator project's context menu or on Visual Studio's project menu. Selecting this item shows Dotfuscator's *Add Project Output* dialog. From this dialog you can browse the other projects in your solution.

Each Visual Studio project type defines a set of output groups that other projects may access. Output groups contain files that are created by the source project. For example, in the screen shot below, the *GettingStarted C#* project defines an output group called **Primary output** and places the output binary in it (**`GettingStarted.exe`**).



The contents of an output group can vary by project configuration. For example, the *GettingStarted* project creates a **`GettingStarted.exe`** for both the Debug and Release configurations, but these are written to different directories when a build is performed. You can tell Dotfuscator to always use a particular output group based on configuration, or you can tell Dotfuscator to always use the output group from whichever project configuration is currently active. When you select the active configuration, Dotfuscator will automatically update its inputs whenever the source project's configuration changes.

Inputs from the File System

To tell Dotfuscator about a package or input assembly on your file system, you add the input via the **Add Input** item that is on the Dotfuscator project's context menu or on Visual Studio's project menu. Selecting this item brings up a file browse dialog that will allow you to choose one or more inputs.

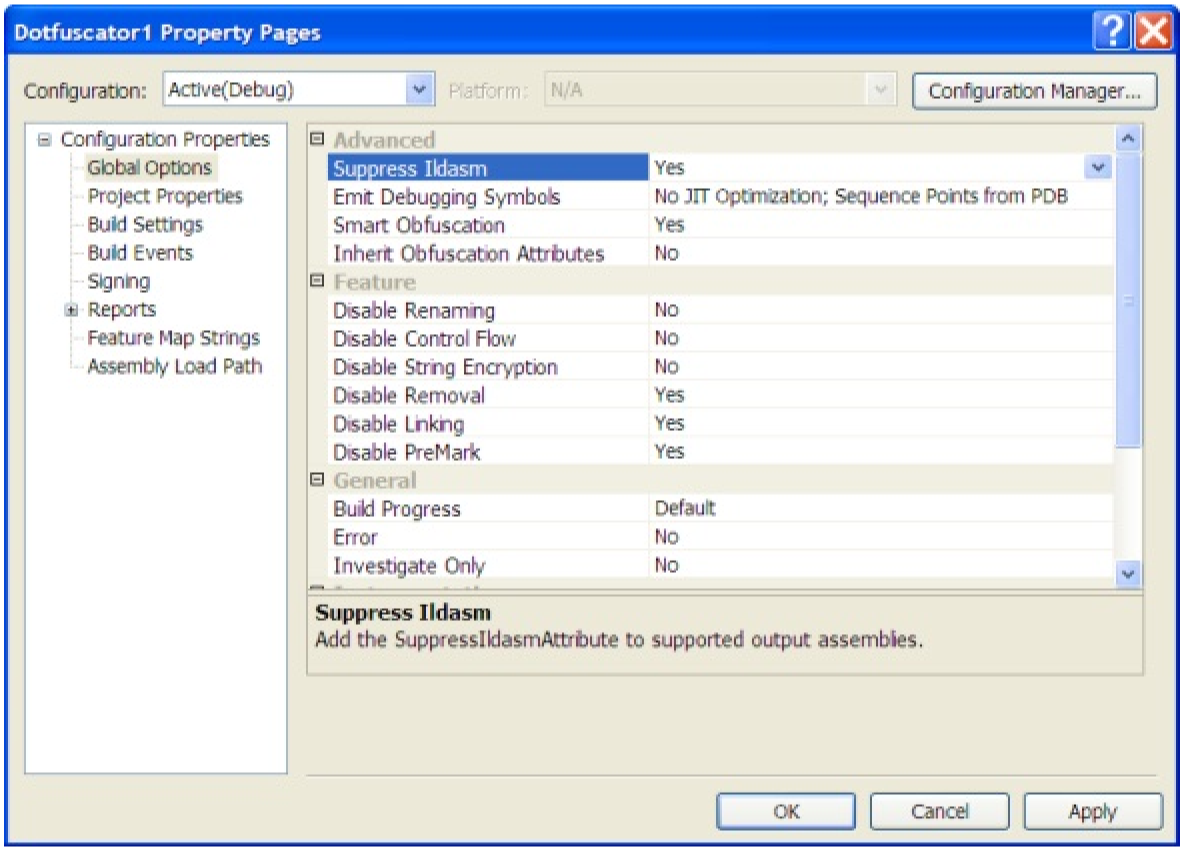
2.5.2.6 Project Properties

In Visual Studio, you can access a Dotfuscator project's property pages from the top level item for the project in Solution Explorer:

Configuration Properties > Global Options

The *Global Options* property page allows you to set the global options for the Project. These are explained in detail in the Global Section. You can also selectively enable Dotfuscator's features. Here is a summary of the properties you can set:

- **Disable** [**feature**]. Dotfuscator allows you to enable or disable each of its transforms. When a transform is disabled, its Configuration Option node appears "grayed out" in solution explorer. You can enable or disable a transform using this property sheet or by right clicking on the appropriate node and checking or un-checking the **Disabled** menu item. The transforms that are Disabled by default in new projects include: Linking, Watermarking, Removal, and String Encryption; Transforms that are Enabled by default in new projects include Renaming, Control Flow and Instrumentation.
- **Build Progress**. This controls the verbosity of Dotfuscator's output during a build.
- **Error**. This setting is for diagnostic purposes. Turning this on will prevent Dotfuscator from deleting the generated IL that is written to the temp directory during a build.
- **Smart Obfuscation**. This allows you to enable or disable automated renaming and removal exclusions for selected application types. See Smart Obfuscation for more details. By default, it is enabled.
- **Investigate Only**. This tells Dotfuscator to generate reports but no output assemblies.
- **Emit Debugging Symbols**. Emit debugging symbols for obfuscated assemblies and control JIT behavior. See Debug Global Option.
- **Suppress Ildasm**. This tells Dotfuscator to add the SuppressIldasmAttribute to all output assemblies which will prevent Microsoft's Ildasm utility from displaying the assembly's IL. This is only valid for assemblies targeting .NET 2.0 and above.
- **Instrumentation**. See Configuring and Running Dotfuscator with Application Analytics.



Advanced	
Suppress Ildasm	Yes
Emit Debugging Symbols	No JIT Optimization; Sequence Points from PDB
Smart Obfuscation	Yes
Inherit Obfuscation Attributes	No
Feature	
Disable Renaming	No
Disable Control Flow	No
Disable String Encryption	No
Disable Removal	Yes
Disable Linking	Yes
Disable PreMark	Yes
General	
Build Progress	Default
Error	No
Investigate Only	No

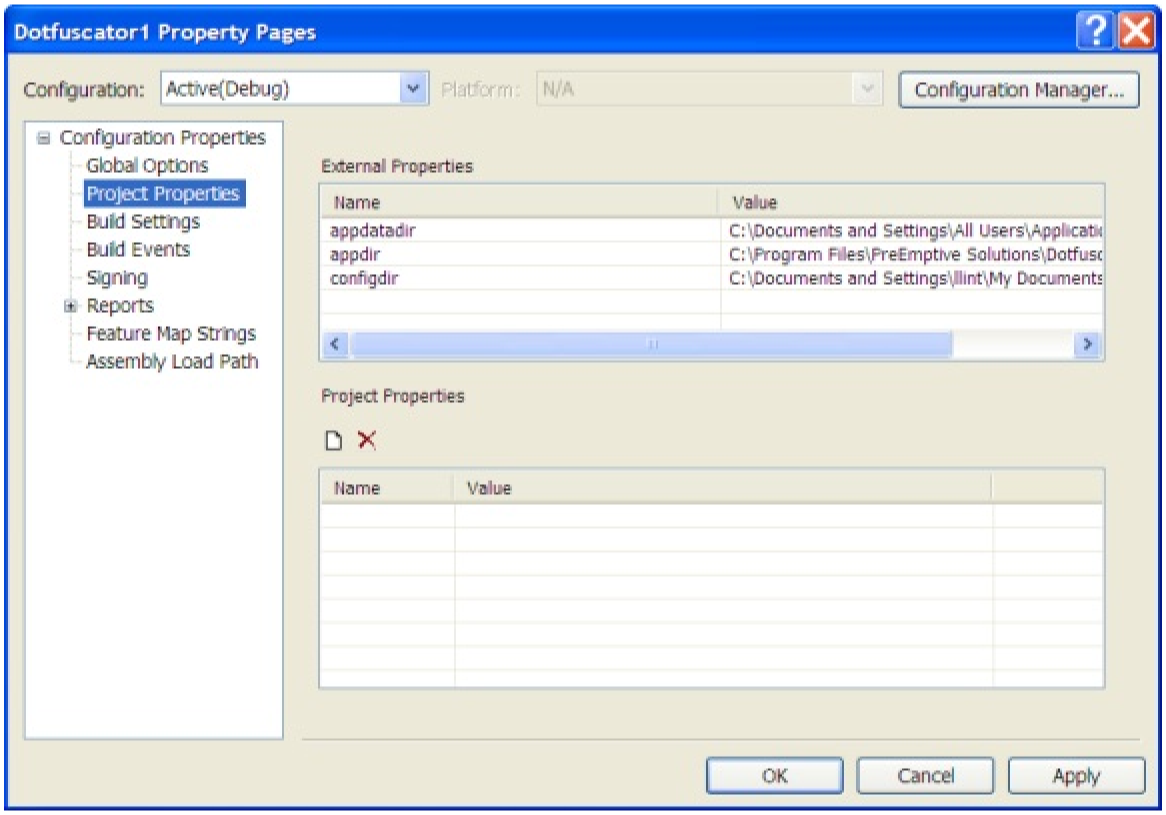
Suppress Ildasm
Add the SuppressIldasmAttribute to supported output assemblies.

Configuration Properties > Project Properties

The Project Properties page contains a facility to view and add user-defined name-value pairs as *Project Properties* and to view *External Properties* that have been defined by Dotfuscator itself.

Project Properties can be thought of as simple string substitution macros that may be used wherever a filename or path is required. See Property List and Properties for a full explanation.

To add a *Project Property*, click the **New** button, then edit the property name and value in the *Project Properties* grid. To delete one or more *Project Properties*, select the row or rows in the *Project Properties* grid that you wish to delete, then click the **Delete** button.



Name	Value
appdatadir	C:\Documents and Settings\All Users\Applicati
appdir	C:\Program Files\PreEmptive Solutions\Dotfusc
configdir	C:\Documents and Settings\lint\My Documents

Configuration Properties > Build Settings

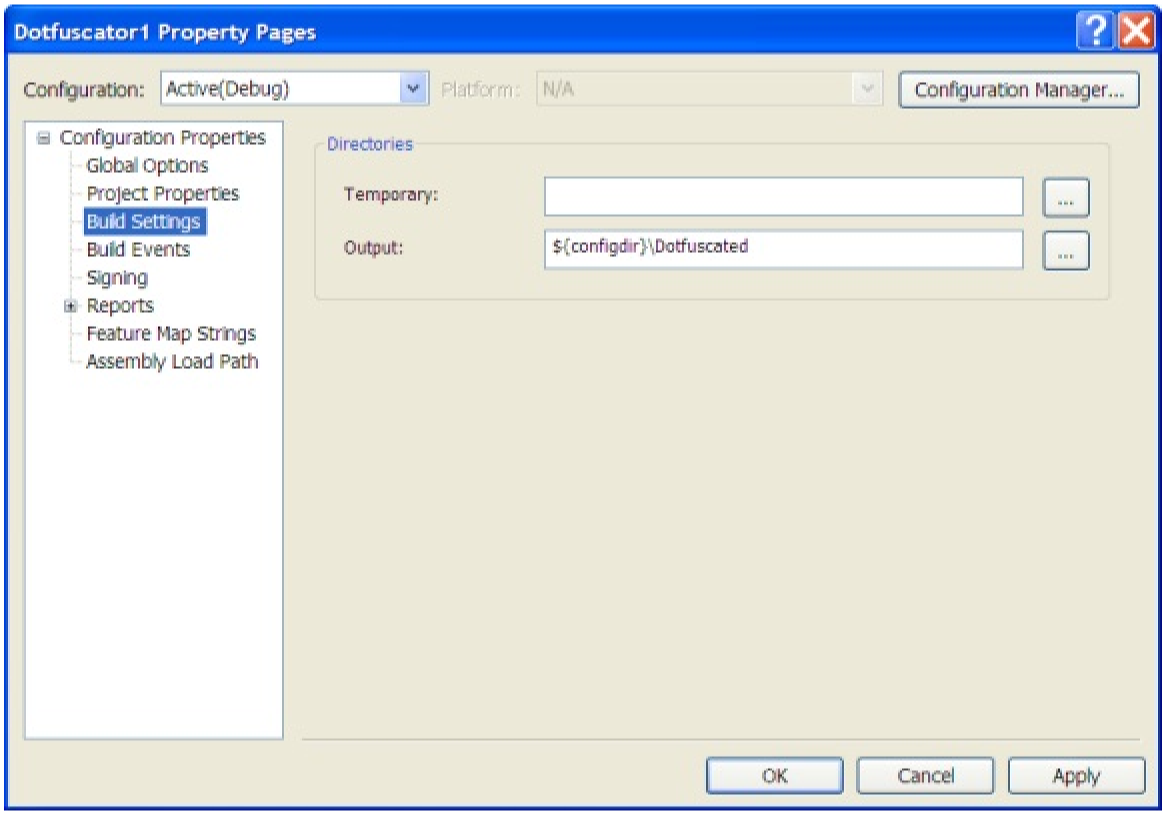
The *Build* property page allows you to specify the output directory and an optional temporary directory.

Build Directories

When you create a new project, the output directory is set by default to **"${configdir}\Dotfuscated"**. The **${configdir}** is a built in property that is expanded to the folder that your project configuration file is saved to.

If you want to choose a different output directory, you can browse for it or type it into the *Output Directory* text box.

You can also optionally specify a temporary directory for Dotfuscator to use for scratch files during processing. If not specified, the temp directory specified by the Windows environment is used.



Temporary:

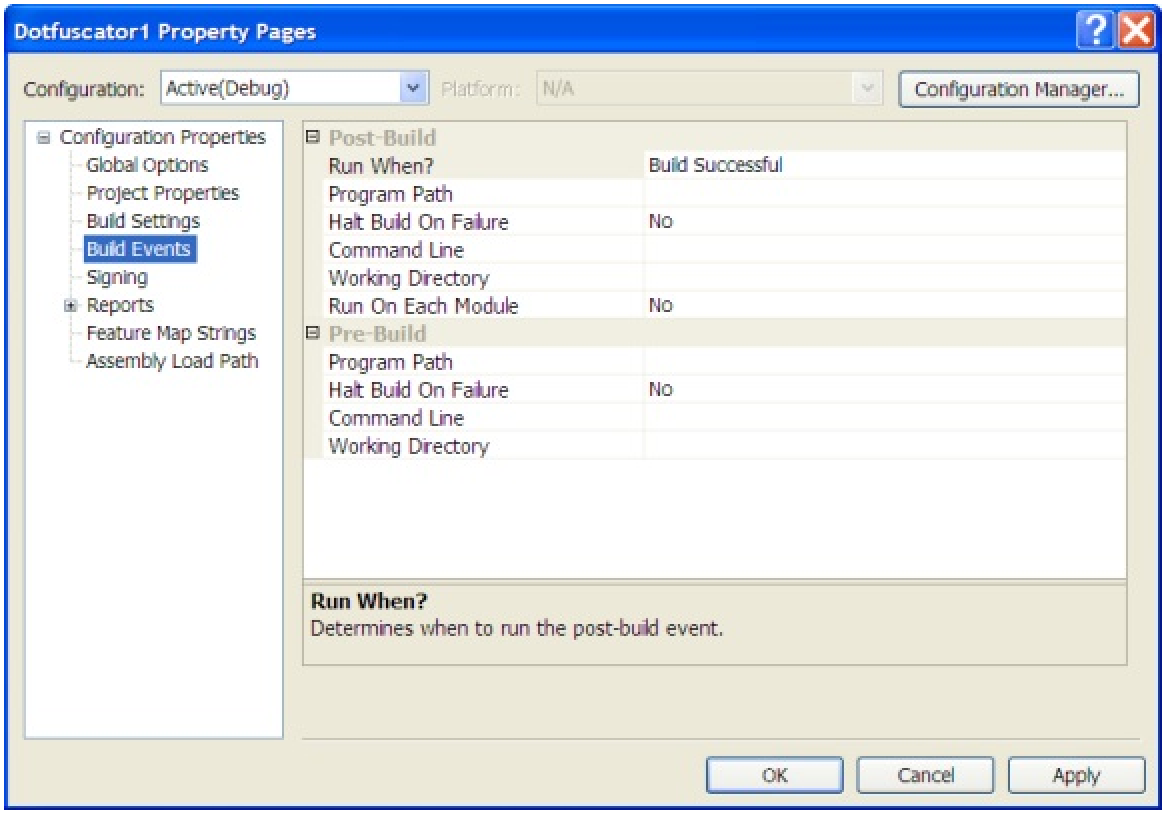
Output: ${configdir}\Dotfuscated

Configuration Properties > Build Events

The *Build Events* property page is where you specify Build Events for your Dotfuscator project. For each event, specify an external program (*Program Path*) that runs when the event occurs. You can specify a working directory and command line options for the program and you can specify if the Dotfuscator build should halt (fail) if the specified program fails.

For the post build event, specify under what conditions it will run (e.g. all the time, only if the build succeeds, or only if the build fails).

You can also specify whether you want the post build event to run only once for the project, or run once for each output module.



Post-Build	
Run When?	Build Successful
Program Path	
Halt Build On Failure	No
Command Line	
Working Directory	
Run On Each Module	No
Pre-Build	
Program Path	
Halt Build On Failure	No
Command Line	
Working Directory	

Run When?
Determines when to run the post-build event.

Configuration Properties > Signing

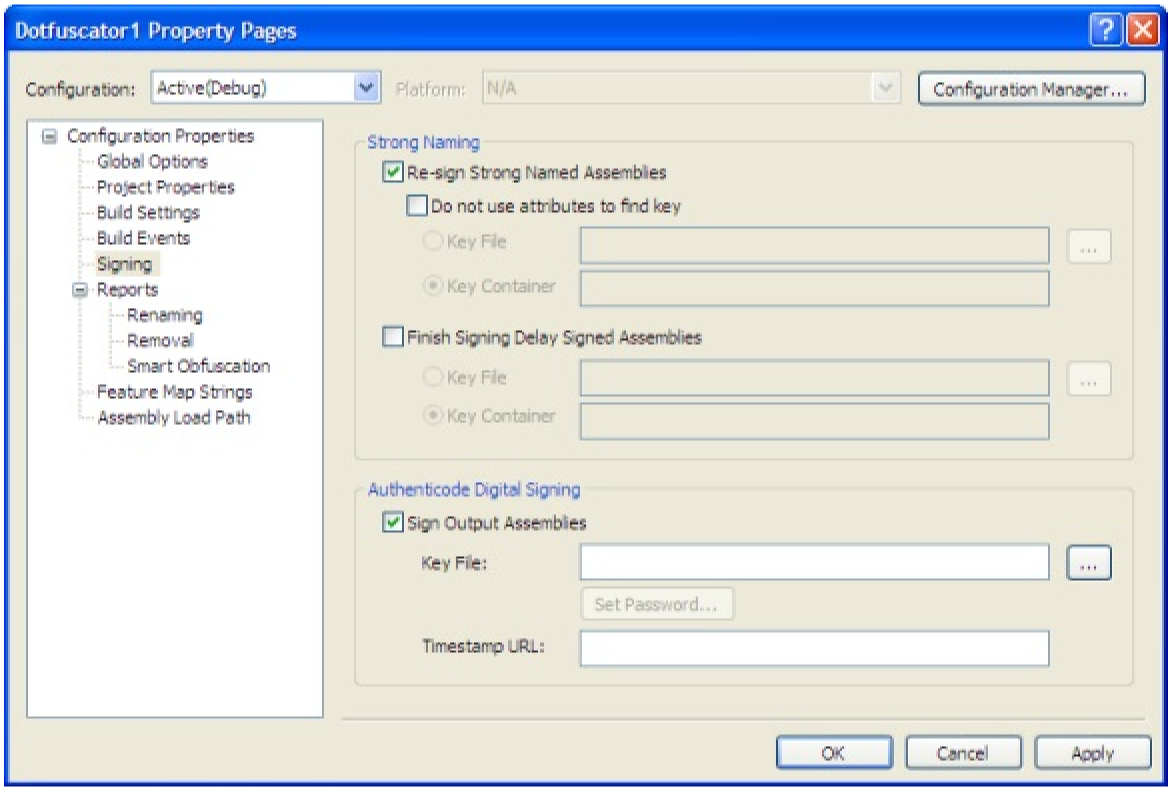
Strong Naming

The *Signing* property page allows you to configure Dotfuscator to automatically sign or resign strongly named assemblies. See Dotfuscating Strong Named Assemblies for more information.

Authenticode Digital Signing

The Authenticode Digital Signing option allows you to attach an Authenticode digital signature to your application. Similar to a security certificate, this signature certifies that the application you are obfuscating and instrumenting is your intellectual property, and allows users to ensure that the resulting binaries were provided by you alone and have not been modified. This feature adds another level of security to safeguard your application. To attach an Authenticode signature to your output assemblies, check the *Sign Output Assemblies* checkbox and then click the **"..."** (ellipsis) to locate your **Key File**, or enter its path in the text box. Once the *Key File* field is properly populated, click the *Set Password...* button to set the password for your Key File.

The **Timestamp URL** field provides the ability for you to specify the URL of an Authenticode timestamp service. This URL will be accessed during Dotfuscator's signing process, and will provide additional data which will allow your assemblies' Authenticode signatures to remain valid after your code-signing certificate has expired. This element is optional. If omitted, this additional data will not be included, and your assemblies' Authenticode signatures will become invalid once your code-signing certificate expires.



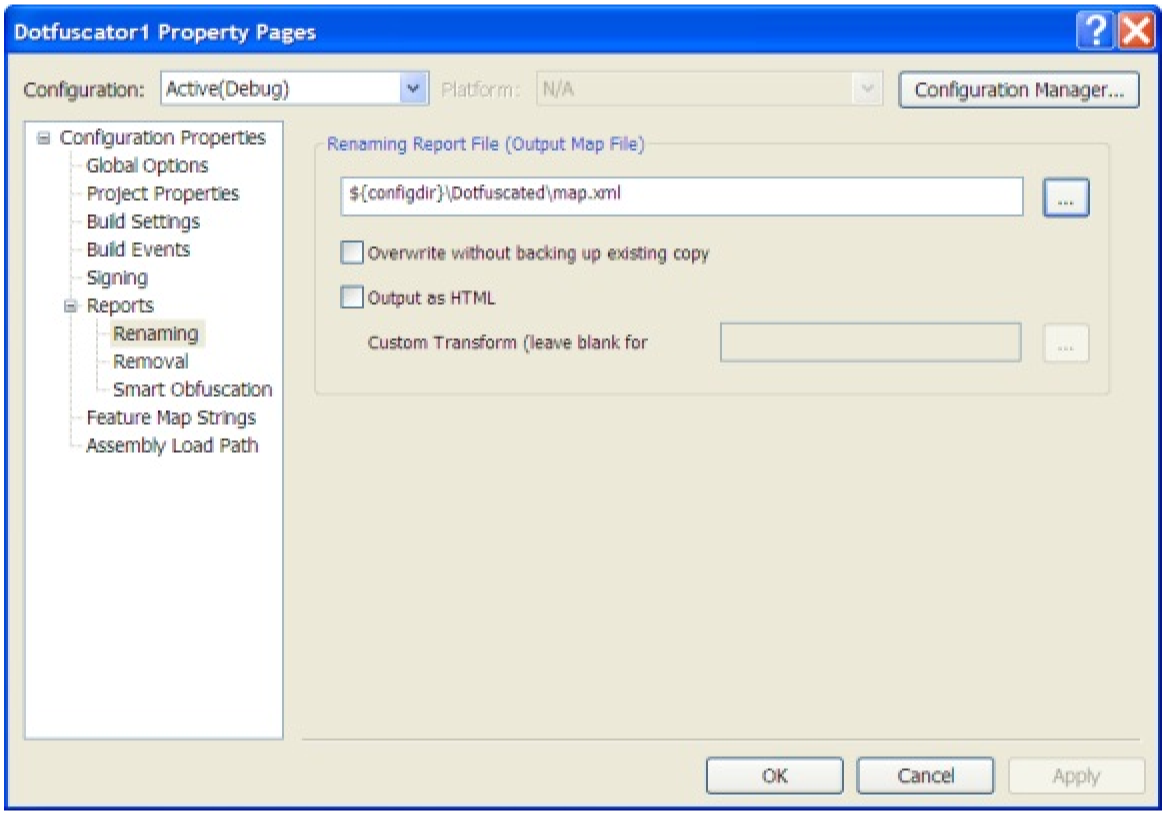
Strong Naming
- [✓] Re-sign Strong Named Assemblies
- [ ] Do not use attributes to find key
- [ ] Finish Signing Delay Signed Assemblies

Authenticode Digital Signing
- [✓] Sign Output Assemblies

Configuration Properties > Reports > Renaming

The renaming report provides a summary of all the elements renamed by Dotfuscator during a specific run, including a statistics section. The *Renaming Report File* (*Output Map File)* section allows you to specify the location to save a renaming map file. You may leave the default value, or enter your preferred path. If you know the name and path of the mapping file you want to use, you can type it directly into the text box. Alternatively, you can browse your file system for the intended file location. The **Browse** button on the right of the text box brings up the *Select Map Output File* window that provides a familiar navigational dialog.

You also have the option of overwriting the output file each time you build the application without generating a backup of the existing copy of the output. Dotfuscator has a default transform that can generate a readable HTML formatted version of the report in addition to the default .XML format. If you do check the *Output As HTML* box, the *Custom Transform (Leave blank for default)* field is activated. As the field name states, leave this blank for the default custom transform or click **"..."** (ellipsis) to select the location of your choice.



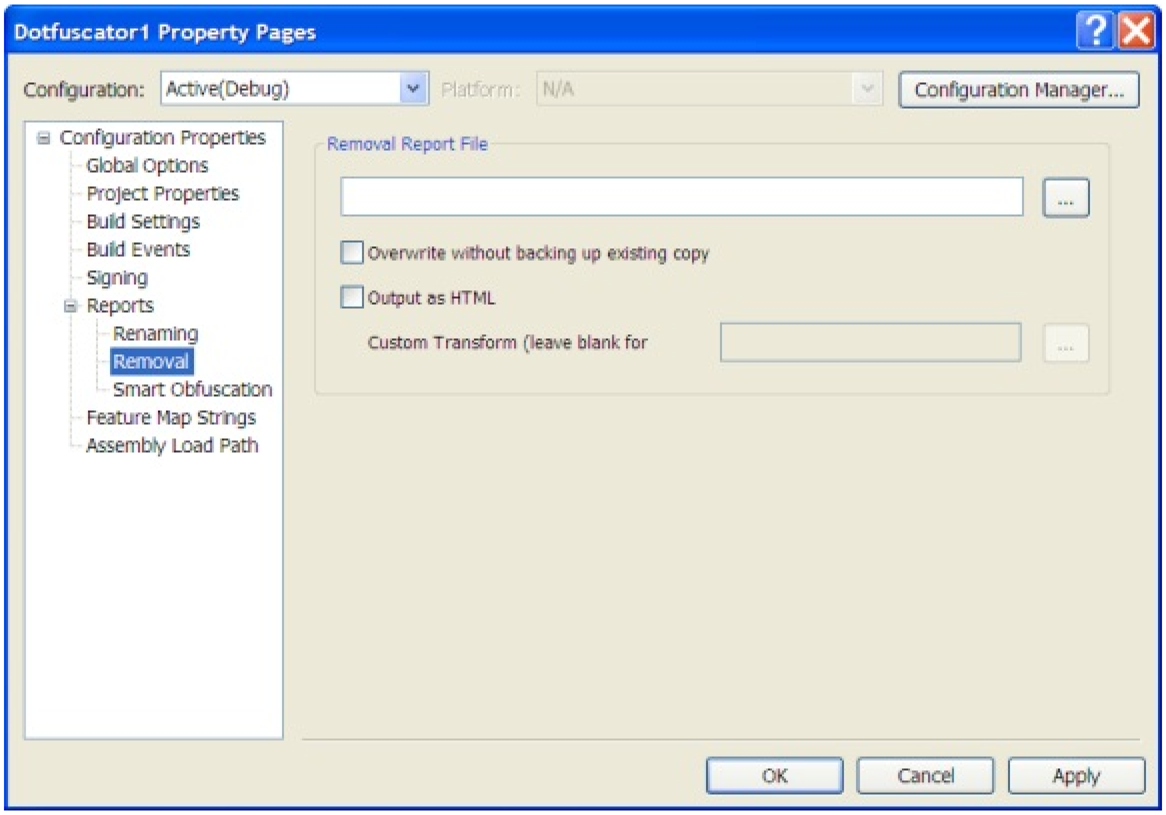
Configuration: Active(Debug)
Platform: N/A

Renaming Report File (Output Map File)
${configdir}\Dotfuscated\map.xml
Overwrite without backing up existing copy
Output as HTML
Custom Transform (leave blank for

Configuration Properties > Reports > Removal

The removal report provides a summary of all the elements removed by Dotfuscator during a specific run, including a statistics section. The *Removal Report File* section allows you to specify the location to save the report. Two approaches are available. If you know the name and path of the file you want to use, you can type it directly into the text box. Alternatively, you can browse your file system for the intended file location. The **"..."** (ellipsis) button on the right of the text box brings up the *Select Removal Report File* window that provides a familiar navigational dialog.

You also have the option of overwriting the output file each time you build the application without generating a backup of the existing copy of the output. The removal report can also be written out as a readable, HTML formatted document, in addition to the XML formatted version. This report provides a quick cross reference that allows you to quickly navigate through all types, fields, and methods to see at a glance which were removed. Checking the *Output as HTML* checkbox tells Dotfuscator to write this report using the same name and path information as the XML version. If the default HTML report doesn't meet your reporting requirements, you can provide your own XSL document that Dotfuscator uses to transform the XML version.If you do check the *Output As HTML* box, the *Custom Transform (Leave blank for default)* field is activated. As the field name states, leave this blank for the default custom transform or click **"..."** (ellipsis) to select the location of your choice.



Removal Report File

☐ Overwrite without backing up existing copy

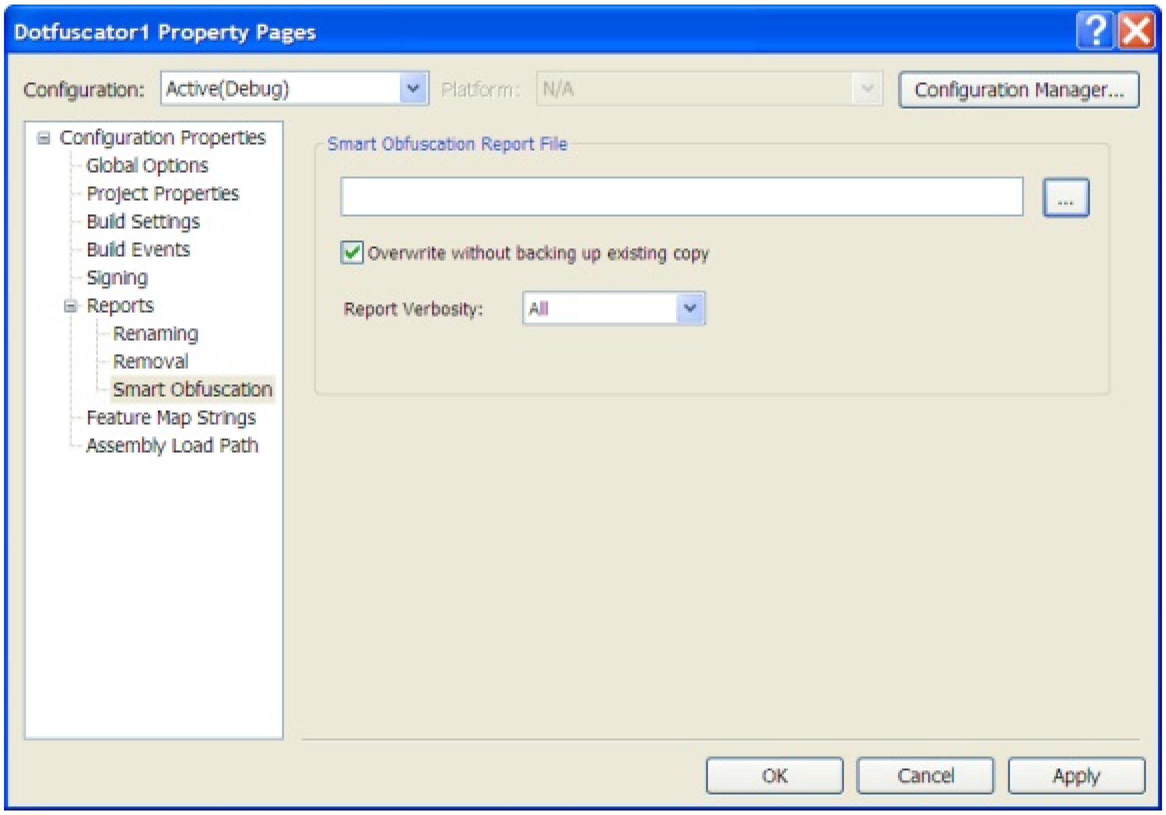
☐ Output as HTML

Custom Transform (leave blank for

Configuration Properties > Reports > Smart Obfuscation

The smart obfuscation report provides a summary of all the elements that could not be renamed or removed based on rules provided by the smart obfuscation feature. If items are excluded by smart obfuscation but no destination for the report is specified, the report will appear in the *Smart Obfuscation Report* view of the Output tab at the bottom of Visual Studio's window. The *Smart Obfuscation Report File* field allows you to specify the location to save the report. Two approaches are available. If you know the name and path of the mapping file you want to use, you can type it directly into the text box. Alternatively, you can browse your file system for the intended file location. The **"..."** (ellipsis) button on the right of the text box brings up the *Select Smart Obfuscation Report File* window that provides a familiar navigational dialog.

You have the option of overwriting the output file each time you build the application without generating a backup of the existing copy of the output. You may also optionally configure the verbosity of the report. Allowed values for the verbosity attribute are **All**, **Warnings Only**, and **None**. The default value is **All**.



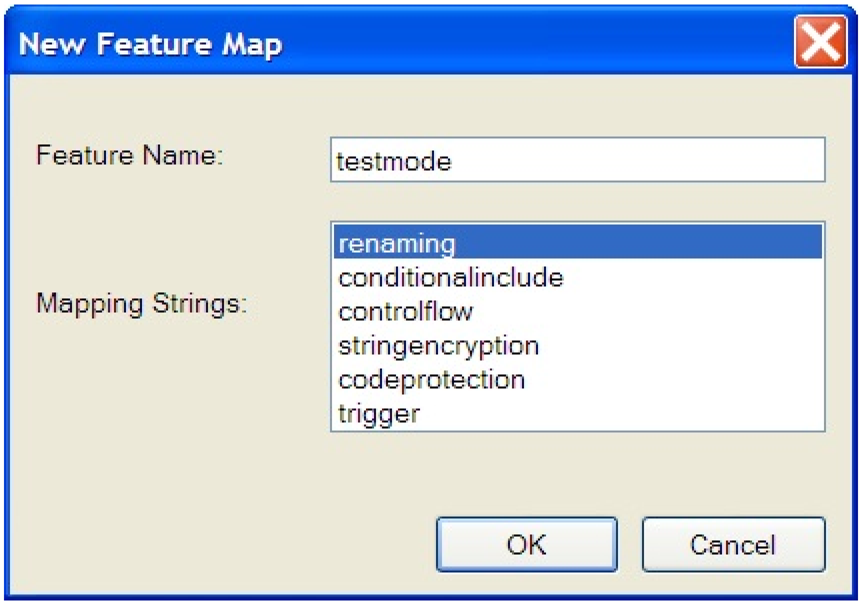
[✓] Overwrite without backing up existing copy

Report Verbosity: All

Configuration Properties > Setup > Feature Map Strings

The feature map strings property page is for Declarative Obfuscation. For a complete description of Dotfuscator's support for Declarative Obfuscation, see Declarative Obfuscation via Custom Attributes. That section describes the *Feature Map* and lists the native feature strings that Dotfuscator understands.

From the toolbar, you can add, edit, and remove feature map strings. The **Add** and **Edit** buttons bring up a dialog that allows you to map feature strings to supported Dotfuscator features.

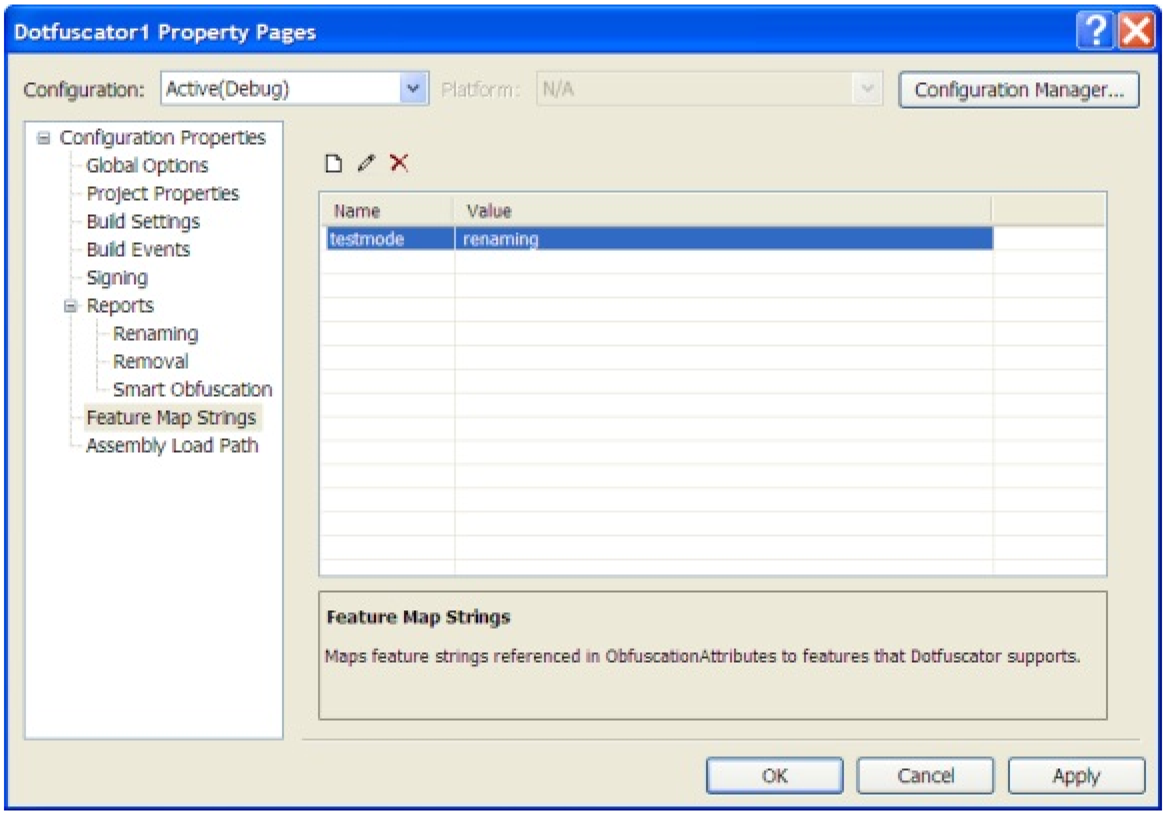


Feature Name: testmode

Mapping Strings:
renaming
conditionalinclude
controlflow
stringencryption
codeprotection
trigger

When you click the Add icon, the New Feature Map dialog box displays. Enter the Feature Name and select the appropriate Mapping String for that feature. To select more than one mapping string, press and hold the Control key and click on the appropriate Mapping String. Click OK when you are done. The new feature and its strings display in the Feature Map Strings Property page:

The new feature and its strings display in the Feature Map Strings Property page:



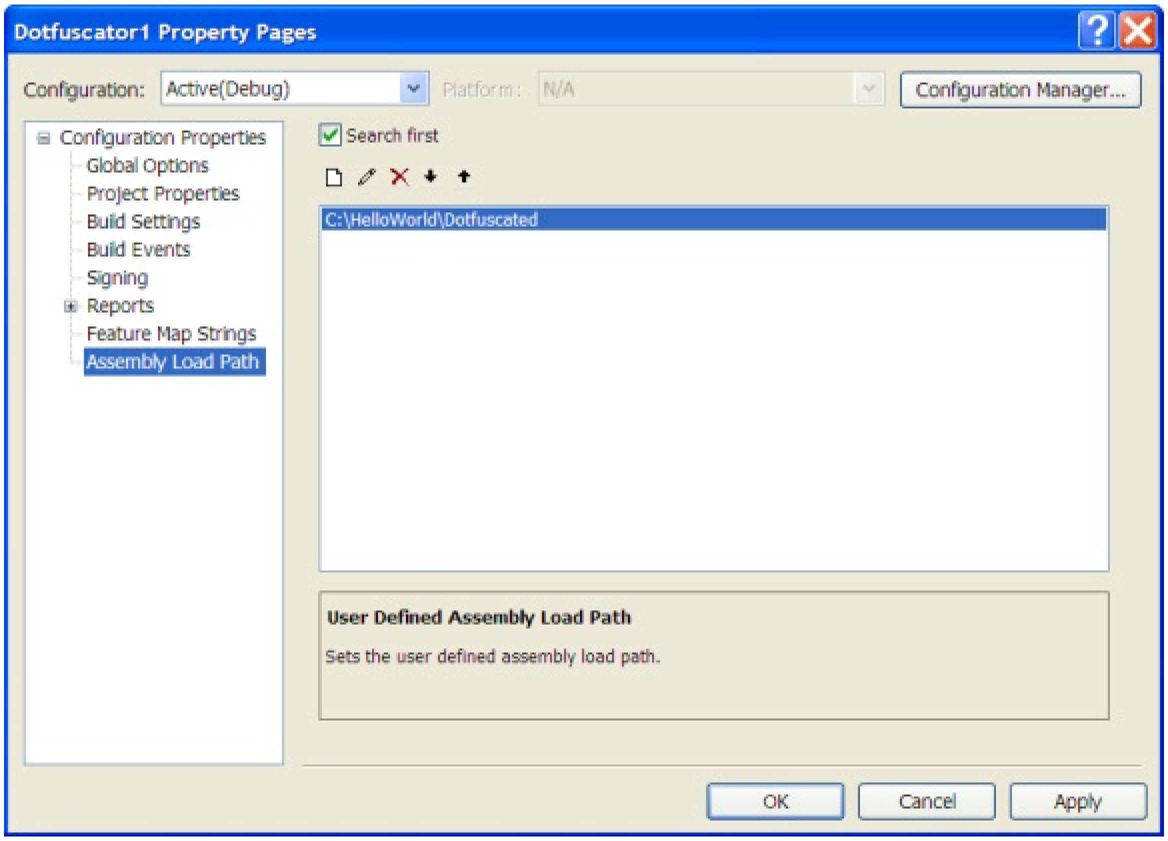
Name	Value
testmode	renaming

Feature Map Strings

Maps feature strings referenced in ObfuscationAttributes to features that Dotfuscator supports.

Configuration Properties > Setup > Assembly Load Path

Using this property page, you can edit your project’s user defined assembly load path. From the toolbar, you can add or remove directories, edit existing directories, or change the order in which they are searched. Check the **Search First** checkbox to have Dotfuscator search the load path before applying its standard search. When unchecked, Dotfuscator will search the loadpath only after applying its standard search.



C:\HelloWorld\Dotfuscated

User Defined Assembly Load Path

Sets the user defined assembly load path.

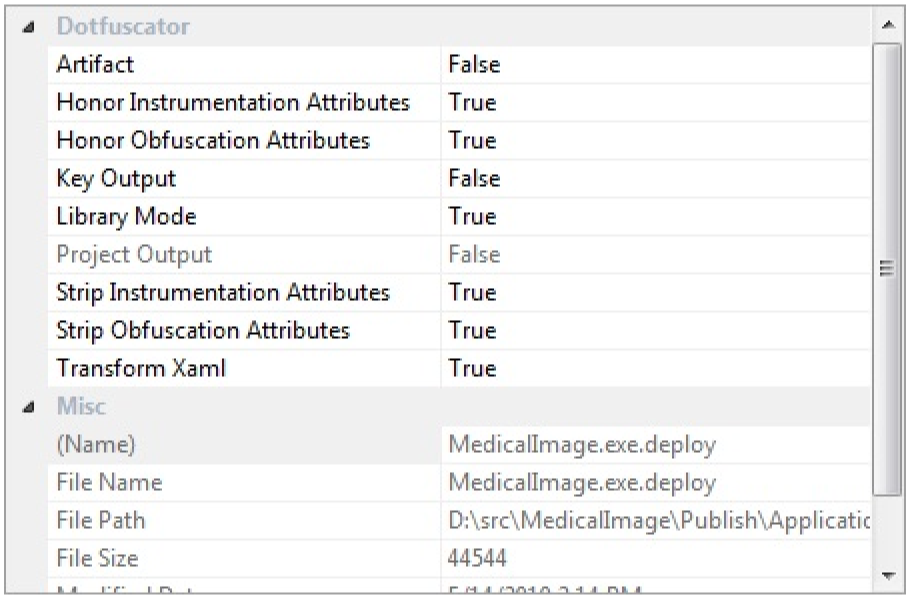
2.5.2.7 Input Assembly Properties

In Visual Studio, every input assembly exposes properties to the properties window. To bring up the properties window for an input assembly, select the input assembly node under your Dotfuscator project in Solution Explorer, then launch Visual Studio's properties window (from the View menu or using F4). You can manage properties for multiple input assemblies by selecting multiple input assemblies; the properties exposed in the properties window apply to the group of selected assemblies.

From here, for each assembly you can set library mode, set the Transform XAML mode, mark if it should be obfuscated or left as a package artifact, and you can configure Declarative Obfuscation via the Honor Obfuscation Attributes and Strip Obfuscation Attributes properties.

You can also set instrumentation options for selected assemblies. Specifically, you can select whether you would like Dotfuscator to honor instrumentation attributes for selected assemblies, and whether you would like Dotfuscator to strip these attributes from the output assembly. See Configuring and Running Dotfuscator with Application Analytics for details.

In addition, there are read-only properties that display information about the input assembly's file.



Dotfuscator	
Artifact	False
Honor Instrumentation Attributes	True
Honor Obfuscation Attributes	True
Key Output	False
Library Mode	True
Project Output	False
Strip Instrumentation Attributes	True
Strip Obfuscation Attributes	True
Transform Xaml	True
Misc	
(Name)	MedicalImage.exe.deploy
File Name	MedicalImage.exe.deploy
File Path	D:\src\MedicalImage\Publish\Applicatic
File Size	44544

In the *Dotfuscator* section of the *Properties* panel, you may set the following to **True** or **False**:

Artifact: Setting the *Artifact* to **True** tells Dotfuscator to not process the specified assembly as an input. It will not be obfuscated or instrumented and all existing signatures will be preserved. Setting the *Artifact* to **False** will cause Dotfuscator to process the assembly as an input for obfuscation and instrumentation.

Honor Instrumentation Attributes: Setting *Honor instrumentation attributes* to **True** tells Dotfuscator to process these attributes and perform the indicated instrumentation transformations on the target assembly. Setting this option to **False** tells Dotfuscator to ignore any instrumentation attributes.

Instrumentation attributes are custom attributes that can be applied in your source code to track application stability, features, usage, and to add shelf life functionality.

Honor Obfuscation Attributes: The default setting of *Honor obfuscation attributes* is **True**, which tells Dotfuscator to process these attributes and perform the indicated obfuscation transformations on the target assembly. Setting this option to **False** tells Dotfuscator to ignore any obfuscation attributes.

Obfuscation attributes are custom attributes that can be applied in your source code to explicitly declare the inclusion or exclusion of types, methods, enums, interfaces, or members from various types of obfuscation. The attribute you would use to include or exclude types, methods, enums, interfaces, and members from obfuscation is **System.Reflection.ObfuscationAttribute**. If you want to denote that a specific assembly will have its items included or excluded from obfuscation, you would use **System.Reflection.ObfuscateAssemblyAttribute**.

Key Output: This setting allows you to explicitly set one of the Dotfuscator project's input assemblies to be the key output assembly. The key output is consumed by deployment projects. If assembly linking is enabled, a linked output assembly will be the key output if one of its source assemblies was marked as the key output. If no input assembly is marked as the key output, Dotfuscator will choose one.

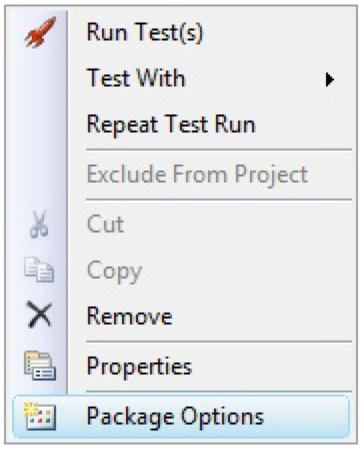
tells Dotfuscator to leave the attributes in the output assembly unless the individual attributes designate that they should be stripped via the StripAfterObfuscation property.

Strip obfuscation Attributes: Dotfuscator can strip out all of the obfuscation attributes when processing is complete, so output assemblies will not contain clues about how it was obfuscated. Selecting this option tells Dotfuscator to remove these attributes from the target output assembly. De-selecting this option tells Dotfuscator to leave the attributes in the output assembly unless the individual attributes designate that they should be stripped via the StripAfterObfuscation property.

Transform XAML: Dotfuscator can rename items in XAML resources as used in Silverlight applications as well as the compiled XAML resources (BAML) of Windows Presentation Foundation applications. The default value is True, which tells Dotfuscator to attempt to rename items in the markup resources and match the renaming to the items references in the code-behind. Leaving this option enabled significantly strengthens the obfuscation of Windows Presentation Foundation and Silverlight applications as well as decreasing the number of items that must be manually excluded from renaming.

2.5.2.8 Input Package Properties

In Visual Studio, some package inputs expose properties to the properties window. To bring up the properties window for an input assembly, select the package node under your Dotfuscator project in Solution Explorer, then right click and select the **Package Options** menu item to launch the package specific properties window .



2.5.2.9 Configuring

Within a Visual Studio Dotfuscator project, you configure settings independently for each feature. Each feature has an editor that may be launched by double clicking on the appropriate item in your Dotfuscator project tree opened in Solution Explorer. Each editor is described in the following topics:

- Renaming Editor

- Control Flow Editor
- String Encryption Editor
- Removal Editor
- Linking Editor
- PreMark Editor
- Instrumentation Editor

2.5.2.10 Building the Project

A Visual Studio Dotfuscator project, like other Visual Studio project types, can be built at the project level and, by default, will be built when its containing solution is built. If you do not want your Dotfuscator project to be built for a particular solution configuration, you can turn off build using Visual Studio's Configuration Manager.

When a Dotfuscator project builds, it applies the settings for its active configuration to the inputs. The output packages and assemblies are then written to the directory specified on the project's *Build* property page.

During the build, Dotfuscator's output is written to Visual Studio's output window, and any build errors are added to the task list.

After a successful build, the output browser is updated. You can activate the output browser by double clicking the **Output Browser** item for your Dotfuscator project in the *Solution Explorer*. The output browser provides a view of your output assemblies similar to that provided by the class browser. It shows original and obfuscated symbol names, as well as indicating which symbols were removed from the input assemblies.

If you chose to generate HTML reports for renaming or removal, these too are available for viewing after a build.

2.5.2.11 The View Menu

The Visual Studio Integrated version of Dotfuscator adds a Dotfuscator cascading menu to Visual Studio's **View** menu. Depending on what you are doing, you will see up to four items on this menu: Two items for viewing reports generated by a successful build, one item for launching the stack trace decoding tool, and one item to generate new shelf life tokens.

Viewing HTML Reports

HTML renaming and removal reports can be viewed in Visual Studio. If the HTML report files exist and your Dotfuscator project is the active project in Solution Explorer, then two menu items (one for each report) are enabled on the **View > Dotfuscator** menu. Clicking on one will bring up the appropriate report in your default browser.

See The Rename Options Tab for setting up HTML Reports for renaming, and The Removal Options Tab for removal.

Stack Trace Decoding Tool

In Visual Studio, Dotfuscator's Stack Trace Decoding Tool is implemented as a Visual Studio tool window. You can activate it by clicking **View > Dotfuscator > Decode Obfuscated Stack Trace** from the menubar. For details on how to use the tool, see Decoding Obfuscated Stack Traces.

Shelf Life Token Generator

In Visual Studio, Dotfuscator's Shelf Life Token generator is implemented as a Visual Studio tool window. You can activate it by clicking **View > Dotfuscator > Generate Shelf Life Token** from the menubar. For details on how to use the tool, see Generate Shelf Life Token.

2.5.2.12 The Help Menu

The Visual Studio Integrated version of Dotfuscator adds a Dotfuscator cascading menu to Visual Studio's **Help** menu. The menu contains the following items:

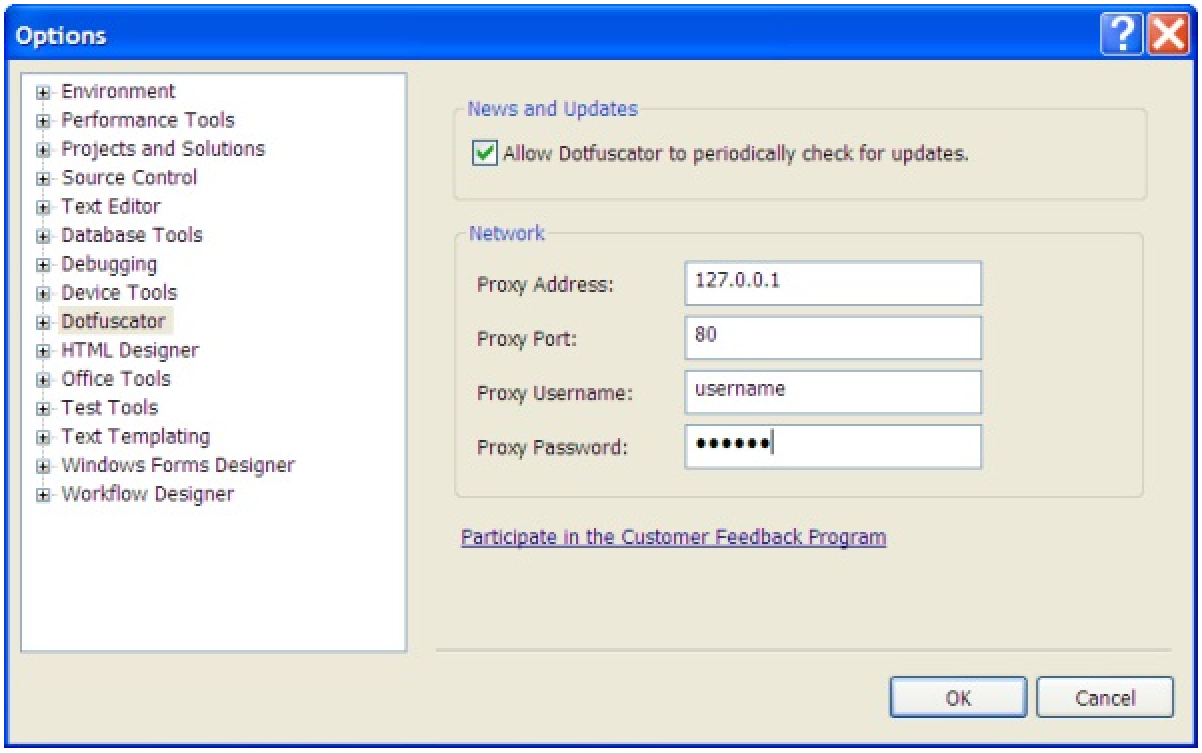
- **Register Product**. This item is only enabled if your copy has not yet been registered. When selected, it invokes Dotfuscator's registration dialog. See Registering and Activating Dotfuscator for more information.
- **Activate Dotfuscator**. This item is only enabled if your Dotfuscator subscription has not been activated, is expired, or is about to expire. See Registering and Activating Dotfuscator for more information.
- **What's New**. When selected, this launches a browser that takes you to Dotfuscator's home page.
- **Customer Feedback Options**. This item allows you to Opt-in to PreEmptive Solutions' anonymous customer feedback program, which enables users to help improve the Dotfuscator family of software products and services.
- **Check For Updates Now**. When selected, Dotfuscator immediately checks the web for updates.
- **About Dotfuscator**. When selected, this brings up Dotfuscator's *About* box which displays user and version information.

2.5.2.13 Set User Preferences

In Visual Studio, Dotfuscator's User Preferences are available by clicking **Tools > Options > Dotfuscator menu**.

In the *News and Updates* section of this dialog box, users may opt to allow Dotfuscator to periodically check for updates.

In the *Network* section of this dialog, the configuration settings of a proxy server for network access may be entered. Proxy information is not required if you do not have a proxy server or if those settings are controlled via Internet Explorer.



News and Updates

☑ Allow Dotfuscator to periodically check for updates.

Network

Proxy Address: 127.0.0.1

Proxy Port: 80

Proxy Username: username

Proxy Password: ••••••

Participate in the Customer Feedback Program

The User Preferences Option window has a link to launch the Customer Feedback Program dialog. The text in the link shows your current opt-in/opt-out status.

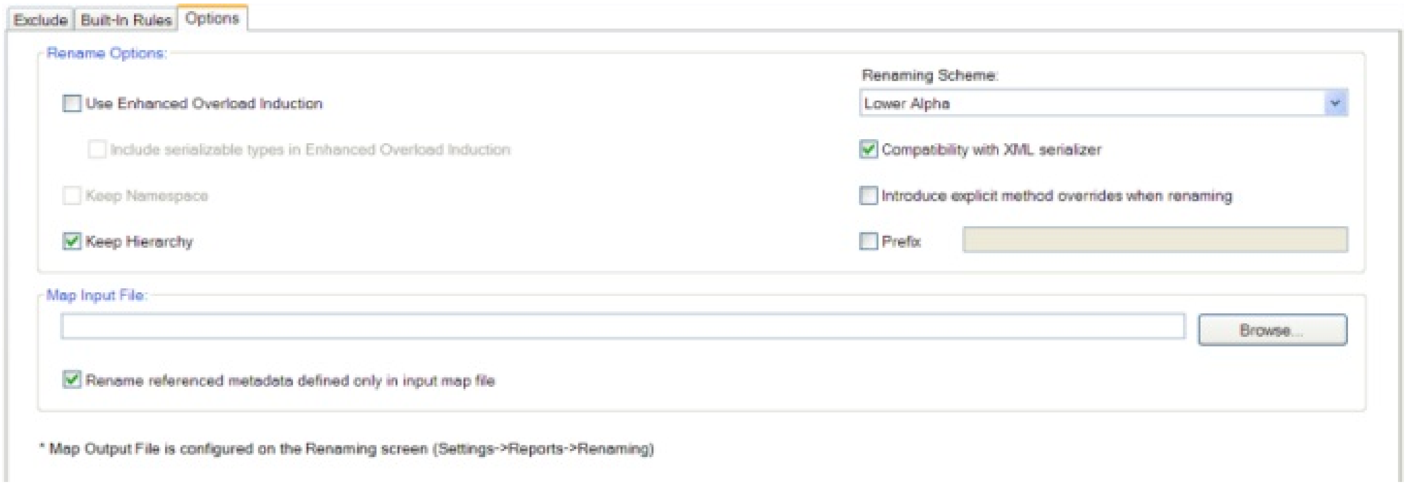
2.5.3 The Renaming Editor

The Renaming editor displays three configuration tabs: the *Exclude* tab, which is used to graphically set custom exclusion rules, and the *Options* tab, which is used to configure other options related to renaming; the *Built-in Rules* tab, which is used to perform common renaming exclusions without having to write custom rules.

For new projects, the default setting for the Renaming transform is **Enabled**.

2.5.3.1 The Rename Options Tab

The Rename *Options* tab is used to set renaming options and identify map files to be used for incremental obfuscation.



- [ ] Use Enhanced Overload Induction
- [ ] Include serializable types in Enhanced Overload Induction
- [ ] Keep Namespace
- [x] Keep Hierarchy

Renaming Scheme: Lower Alpha

- [x] Compatibility with XML serializer
- [ ] Introduce explicit method overrides when renaming
- [ ] Prefix

- [x] Rename referenced metadata defined only in input map file

* Map Output File is configured on the Renaming screen (Settings->Reports->Renaming)

The Renaming *Option* section contains three checkboxes used to select or deselect the Enhanced Overload Induction, Keep Namespace, and Keep Hierarchy options.

You can select your preferred **Renaming Scheme** from the dropdown list. The predefined renaming schemes are described in Renaming Schemes.

Checking the **Compatibility with XML serializer** checkbox adds additional rules to the renamer. All classes and members will be renamed in a way that allows XML Serialization. See XML Serialization and Renaming.

Checking the **Introduce explicit method overrides when renaming** checkbox allows overriding methods to have different names from those of the methods they override.

You can select a rename prefix by checking the **Rename Prefix** checkbox. The text that you enter in the textbox is added to the beginning of every obfuscated type name. If you do not enter any text, Dotfuscator selects a prefix for each input module, based on the module name. See Renaming Prefixes for more information about renaming prefixes.

The **Map Input File** section allows you to specify a map file from a previous run so that the naming scheme is retained across successive Dotfuscator runs, a process known as Incremental Obfuscation. Specific details of this feature are presented in the Incremental Obfuscation section. Two approaches are available. If you know the name and path of the mapping file you want to use, you can type it directly into the edit box. Alternatively, you can browse your file system for the intended file location. The **Browse** button on the right of the edit box brings up the *Select Map Input File* window that provides a familiar navigational dialog.

For ease of navigation, this screen has a note marked with an asterisk (*) at the bottom reminding you that the Map Output File is configured on the Renaming Reports Tab located in the Settings Tab.

2.5.3.1.1 Enhanced Overload-Induction Method Renaming

Dotfuscator extends Overload-Induction™ by allowing a method's return type or a field's type to be used as a criterion in determining method or field uniqueness. This feature allows up to 15% more redundancy in method and field renames. In addition, since overloading on method return type or field type is not allowed in source languages such as C# and VB, this further hinders decompilers.

This feature also relies on compile-time analysis of your application. Therefore, remote method calls cannot use this feature because remoting throws an ambiguous match exception when calling a method on a remote object that differs only by the return type from another method of the type. Therefore, when using remoting, you have two options. First, do not use enhanced overload induction; normal overload induction will still occurs and is perfectly safe. Second, exclude remotely called classes from renaming. Because of the risks involved in using this option with remoting, this feature is turned off by default.

For similar reasons, Enhanced Overload Induction is automatically suppressed on all types marked as serializable. If this is not the desired behavior, then the default can be changed by adding the "**`enhancedOIOnSerializables`**" option to the renaming section of the configuration file (see renaming options), or by checking the *Include serializable types in Enhanced Overload Induction* box on the *Renaming Options* tab in the user interface.

2.5.3.1.2 Class Renaming Options

Full Class Renaming

The default methodology renames the class and namespace name to a new, smaller name. The idea behind this is quite simple. Our example becomes:

Original Name	New Name
`Preemptive.Application.Main`	`a`
`Preemptive.Application.LoadData`	`b`
`Preemptive.Tools.BinaryTree`	`c`
`Preemptive.Tools.LinkedList`	`d`

Note: All classes are now in the `<global>` namespace.

Keepnamespace Option

This methodology is excellent as a way to hide the names of your classes while maintaining namespace hierarchy. You give up some size reduction and obfuscation potential, but preserve your namespace names. This is useful for libraries that may be linked to obfuscated code, or for applications that use already obfuscated code. An example of this type of renaming is:

Original Name	New Name
`Preemptive.Application.Main`	`Preemptive.Application.a`

`Preemptive.Application.LoadData`	`Preemptive.Application.b`
`Preemptive.Tools.BinaryTree`	`Preemptive.Tools.a`
`Preemptive.Tools.LinkedList`	`Preemptive.Tools.b`

Note: `Keepnamespace` and `Keephierarchy` are mutually exclusive options.

Keephierarchy Option

This option tells Dotfuscator to preserve the namespace hierarchy, while renaming the namespace and class names. For example:

Original Name	New Name
`Preemptive.Application.Main`	`a.a.a`
`Preemptive.Application.LoadData`	`a.a.b`
`Preemptive.Tools.BinaryTree`	`a.b.a`
`Preemptive.Tools.LinkedList`	`a.b.b`

Note: `KeepNamespace` and `KeepHierarchy` are mutually exclusive options.

Renaming Prefixes

In some cases it is desirable to have unique top level type names across assemblies, even if those type names are not visible outside their defining assemblies. This is done by running all assemblies through Dotfuscator at the same time. As this approach is not always feasible, Dotfuscator provides a way to enforce uniqueness even across runs, using a renaming prefix.

Renaming prefixes are appended to top level renamed type names. You can specify your own renaming prefix that will be used for all assemblies in a given Dotfuscator run, or you can allow Dotfuscator to pick a prefix for you, based on the type's module name.

One interesting application of this feature is namespace induction. By defining a custom renaming prefix that ends with a "**.**", (e.g. "**MY_PREFIX.**"), you can place your obfuscated types in a custom defined, common namespace.

Examples:

Original Name	Prefix	Type Renaming	New Name
`Application.Main`	`[default]`	`default`	`MyApplicationa`
`Application.LoadData`	`myprefix`	`default`	`myprefixa`
`Tools.BinaryTree`	`myprefix`	`keephierarchy`	`a.myprefixa`
`Tools.LinkedList`	`myprefix`	`keepnamespace`	`Tools.myprefixa`
`Tools.Proxy`	`mynamespace.`	`keepnamespace`	`Tools.mynamespace.a`

There are three ways you can configure renaming prefixes for your project:

- Using the GUI, check the **Rename Prefix** checkbox on the **Rename Options** subtab. If you want to manually specify the prefix, type it in the text box; otherwise leave it blank to have Dotfuscator generate the prefix.
- On the command line, use the **`/prefix:[on > off]`** option. To specify a custom prefix from the command line, use the **`/p`** option to define a property named "**`prefix`**".

Prefix On

```
Dotfuscator /p=prefix=MY_PREFIX /pref:on [other options...]
```

- Using a text or XML editor, manually add an option called "**`prefix`**" to the renaming section. To define a custom prefix, add a "**`prefix`**" property in the **`propertylist`** section, with your custom string.

Define a Custom Prefix

```
<propertylist>
   <!-- defining prefix here tells the renamer to use the value as the
      renaming prefix, if renaming prefix is enabled -->
   <property name="prefix" value="MY_PREFIX"/>
</propertylist>
<renaming>
   <!-- this turns on the renaming prefix feature -->
   <option>prefix</option>
...
</renaming>
```

2.5.3.1.3 XML Serialization and Renaming

Dotfuscator allows you to globally switch the renaming algorithm to rename classes and members in a way that is compliant with the XML Serializer. If you intend for fully obfuscated classes to be serialized by your application, then run the renamer in this mode; this mode is not required if you are excluding serializable classes from renaming.

Here is a list of rules the renamer follows when running in this mode:

- Enhanced Overload Induction is turned off.
- Public instance properties and fields within all types are given names that are unique up and down the inheritance hierarchy.
- Property metadata is preserved on properties decorated with any attribute in the **`System.Xml.Serialization`** namespace.
- On public types that implement the **`System.Collections.ICollection`** interface, the **`Add`** method is excluded from renaming and Property metadata is preserved on the **`Item`** property.

- On public types that implement the **`System.Collections.IEnumerable`** interface, the **`Add`** and **`GetEnumerator`** methods are excluded from renaming.

See renaming options for the configuration option that sets this mode. From the user interface, you can set this mode in the Renaming Options Tab.

2.5.3.1.4 Introduce Explicit Method Overrides When Renaming

This functionality lets Dotfuscator rename more methods by allowing it to introduce explicit (i.e. non-syntactic) method overrides. In other words, overridden methods can have different names than the methods they override. For example, ordinarily, if a method overrides Object.ToString(), Dotfuscator would not be able to rename it without breaking the override relationship, since typically the Object class is not in an input assembly and therefore its ToString() would not be renamed. With this setting, Dotfuscator can rename the overriding method and introduce metadata that tells the CLR that the method is meant to override Object.ToString().

Original Name	New Name
`System.Object.ToString()`	`ToString() [class not in input assembly]`
`Preemptive.MyClass.ToString()`	`a()`

2.5.3.2 The Rename Exclude Tab

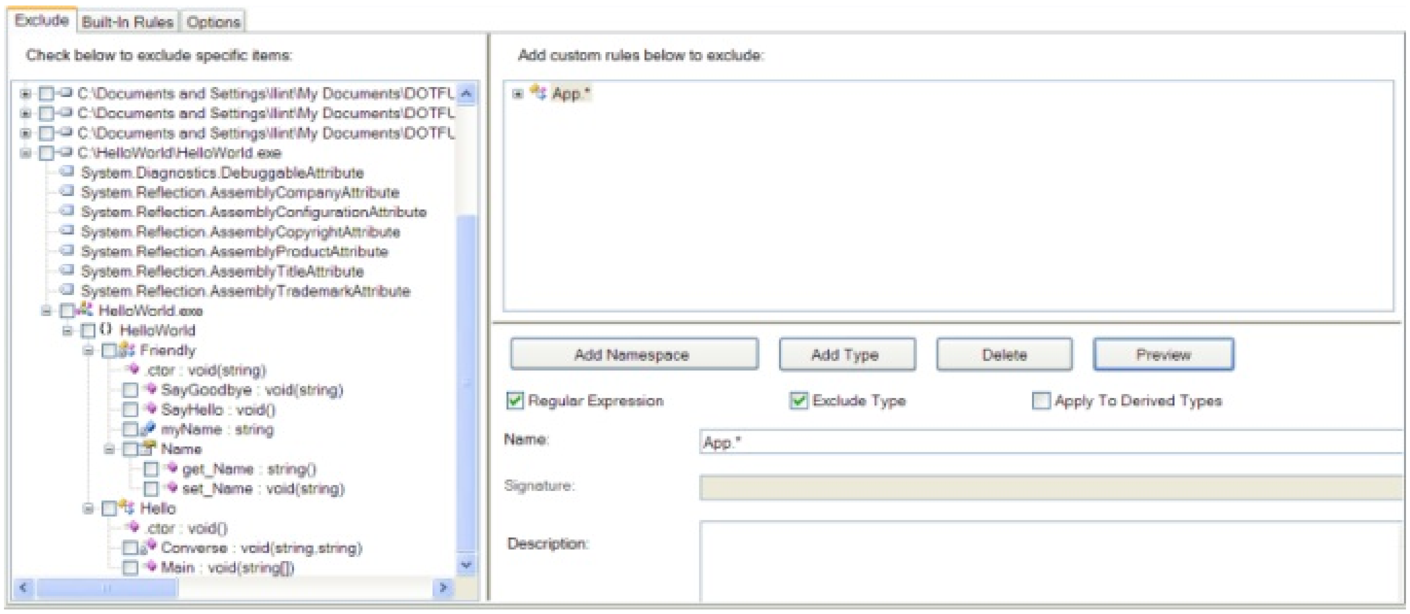
The Rename *Exclude* Tab gives you complete granular control over all parts of your program that you may wish to exclude from the renaming process.

You may exclude specific items from renaming by browsing the tree view of your application and checking the items you want to exclude. In addition, you may visually create your own custom rules for selecting multiple items for exclusion.

To help you fine-tune your exclusion rules, you can preview their effects at any time. The application tree view shades all items selected for exclusion. You can preview the cumulative effects of all rules, or preview a specific rule that you select.

See the section on Using the Graphical Rules Editing Interface for detailed information about working with Inclusion and Exclusion Rules.

See the section on Renaming Exclusion Rules for a detailed discussion of renaming exclusion rules.


Exclude
Built-In Rules
Options
Check below to exclude specific items:
Add custom rules below to exclude:
Add Namespace
Add Type
Delete
Preview

```
C:\Documents and Settings\llint\My Documents\DOTFU
C:\Documents and Settings\llint\My Documents\DOTFU
C:\Documents and Settings\llint\My Documents\DOTFU
C:\HelloWorld\HelloWorld.exe
    System.Diagnostics.DebuggableAttribute
    System.Reflection.AssemblyCompanyAttribute
    System.Reflection.AssemblyConfigurationAttribute
    System.Reflection.AssemblyCopyrightAttribute
    System.Reflection.AssemblyProductAttribute
    System.Reflection.AssemblyTitleAttribute
    System.Reflection.AssemblyTrademarkAttribute
  HelloWorld.exe
    {} HelloWorld
      Friendly
        .ctor : void(string)
        SayGoodbye : void(string)
        SayHello : void()
        myName : string
        Name
          get_Name : string()
          set_Name : void(string)
      Hello
        .ctor : void()
        Converse : void(string,string)
        Main : void(string[])
```

```
App.*
```

[✓] Regular Expression [✓] Exclude Type [ ] Apply To Derived Types

Name: App.*

Signature:

Description:

2.5.3.2.1 Renaming Exclusions

The *exclude list* section provides a dynamic way to fine tune the renaming of the input assemblies. The user specifies a list of rules that are applied at runtime. If a rule selects a given class, method, or field, then that item is not renamed.

These rules are applied *in addition to* rules implied by global options such as library.

Rules are logically **OR**-ed together.

Regular Expressions (REs) may be used to select namespaces, types, methods or fields. The optional **regex** attribute is used for this purpose. The default value of **regex** is false. If **regex** is true then the value of the **name** attribute is interpreted as a regular expression; if it is false, the name is interpreted literally. This is important since regular expressions assign special meaning to certain characters, such as the period. Here are some examples of simple regular expressions:

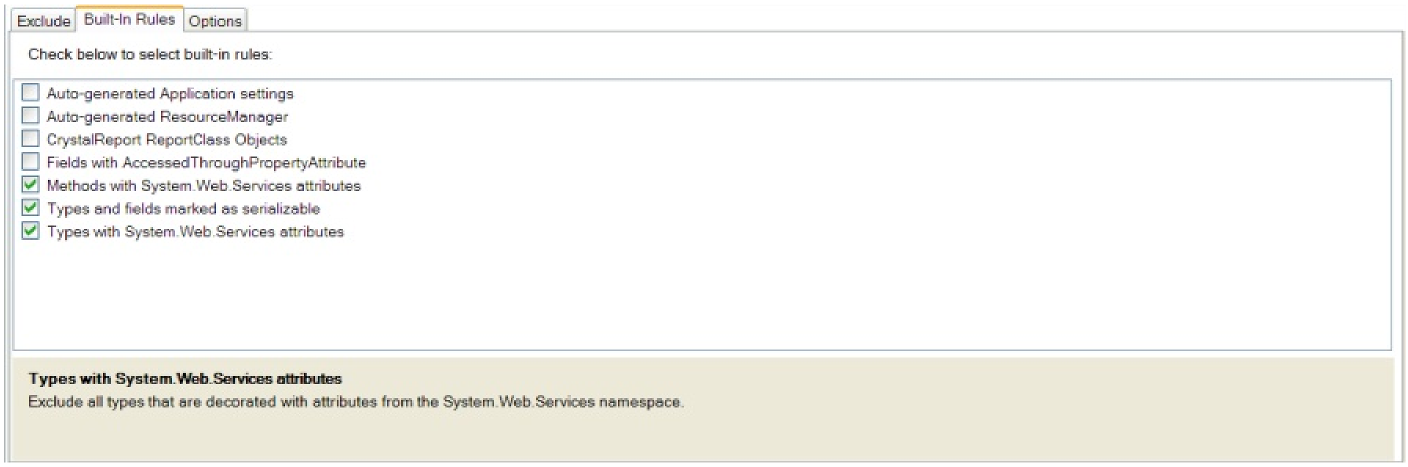
Here are some examples of simple regular expressions:

```
.*                  Matches anything
MyLibrar.           Matches MyLibrary, MyLibrari, etc.
My[\.]Test[\.]I.*   Matches My.Test.Int1,My.Test.Internal, etc.
Get.*               Matches GetInt, GetValue, etc.
Get*                Matches Ge,Get,Gett,Gettt, etc.
```

Please refer to the .NET Framework documentation for a full description of the regular expression syntax.

2.5.3.3 The Rename Built-In Rules Tab

Dotfuscator's renaming *Built-In Rules* tab displays renaming exclusion rules defined in **%ProgramData%\PreEmptive Solutions\Common\dotfuscatorReferenceRule_v1.4.xml**. These rules are standard exclusions that apply to specific application types or technologies. Each rule has a description that displays on the form when the rule is selected. You can apply a built-in rule to your project by checking its checkbox.



Check below to select built-in rules:

- [ ] Auto-generated Application settings
- [ ] Auto-generated ResourceManager
- [ ] CrystalReport ReportClass Objects
- [ ] Fields with AccessedThroughPropertyAttribute
- [✓] Methods with System.Web.Services attributes
- [✓] Types and fields marked as serializable
- [✓] Types with System.Web.Services attributes

Types with System.Web.Services attributes
Exclude all types that are decorated with attributes from the System.Web.Services namespace.

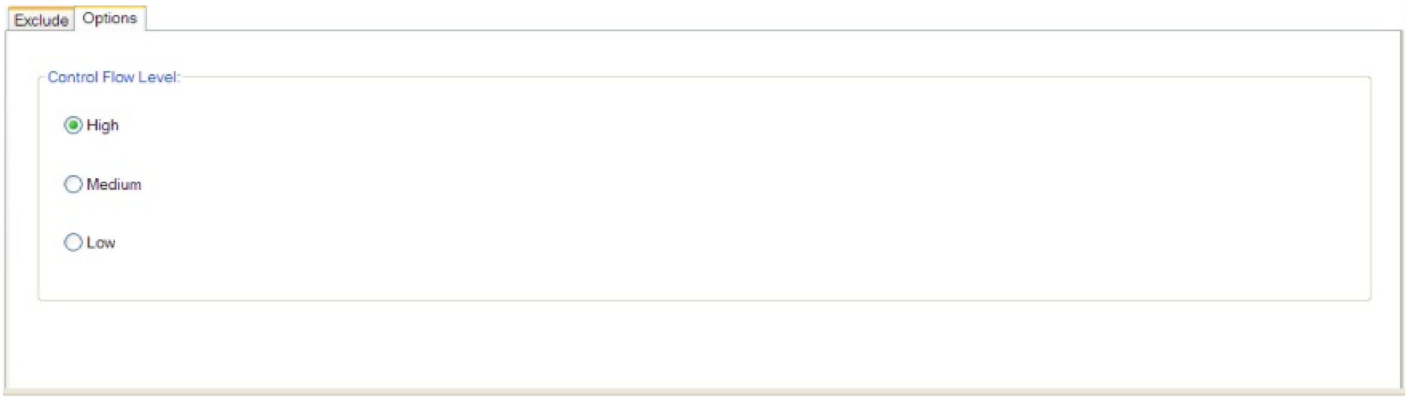
2.5.4 The Control Flow Editor

The *Control Flow* editor displays two configuration tabs: the *Exclude* tab, which is used to graphically set custom exclusion rules, and the *Options* tab, which is used to configure other options related to control flow obfuscation.

For new projects, the default setting for the Control Flow transform is **Enabled**.

2.5.4.1 The Control Flow Options Tab

The *Control Flow Options* tab is used to set the global level of control flow obfuscation.



Control Flow Level:

[✓ High]

Medium

Low

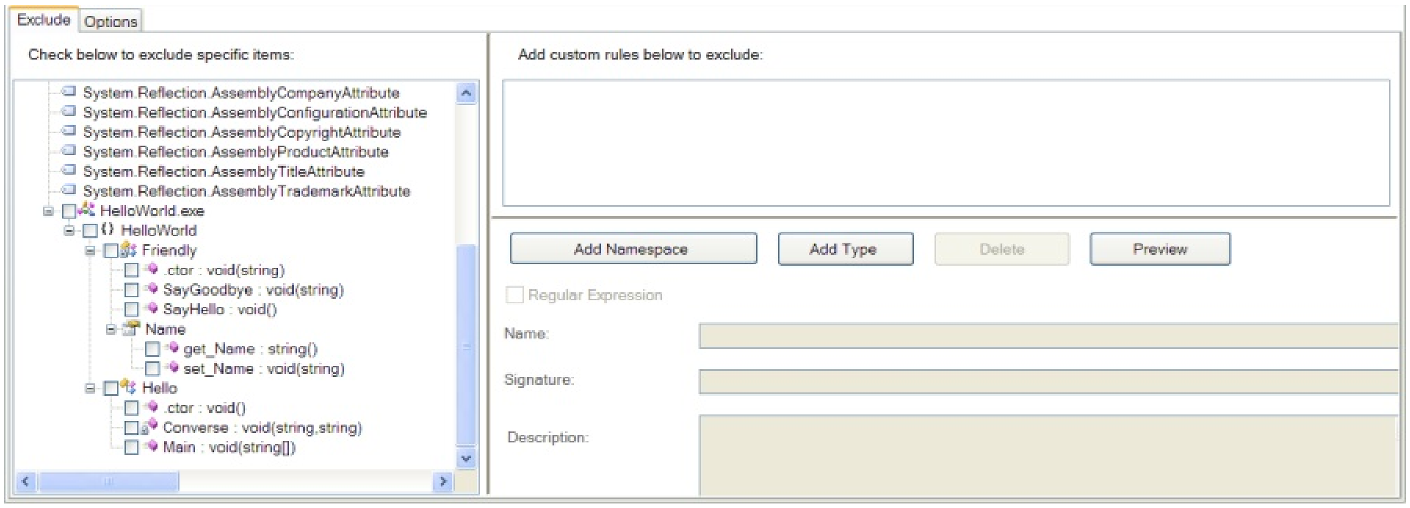
2.5.4.2 The Control Flow Exclude Tab

The *Control Flow Exclude* Tab gives you complete granular control over all parts of your program that you may wish to exclude from the control flow obfuscation process. From here, you can also disable control flow obfuscation altogether.

You may exclude specific items from control flow obfuscation by browsing the tree view of your application and checking the items you want to exclude. In addition, you may visually create your own custom rules for selecting multiple items for exclusion.

To help you fine-tune your exclusion rules, you can preview their effects at any time. The application tree view will shade all items selected for exclusion. You can preview the cumulative effects of all rules, or preview a specific rule that you select.

See the section on The Rules Editing Interface for detailed information about working with Inclusion and Exclusion Rules.



Check below to exclude specific items:

```
System.Reflection.AssemblyCompanyAttribute
System.Reflection.AssemblyConfigurationAttribute
System.Reflection.AssemblyCopyrightAttribute
System.Reflection.AssemblyProductAttribute
System.Reflection.AssemblyTitleAttribute
System.Reflection.AssemblyTrademarkAttribute
HelloWorld.exe
  HelloWorld
    Friendly
      .ctor : void(string)
      SayGoodbye : void(string)
      SayHello : void()
      Name
        get_Name : string()
        set_Name : void(string)
    Hello
      .ctor : void()
      Converse : void(string,string)
      Main : void(string[])
```

Add custom rules below to exclude:

Regular Expression

Name:

Signature:

Description:

2.5.5 The String Encryption Editor

The *String Encryption* editor displays only one configuration tab: the *Include* tab, which is used to graphically set custom inclusion rules for string encryption.

For new projects, the default setting for the String Encryption transform is **Disabled**.

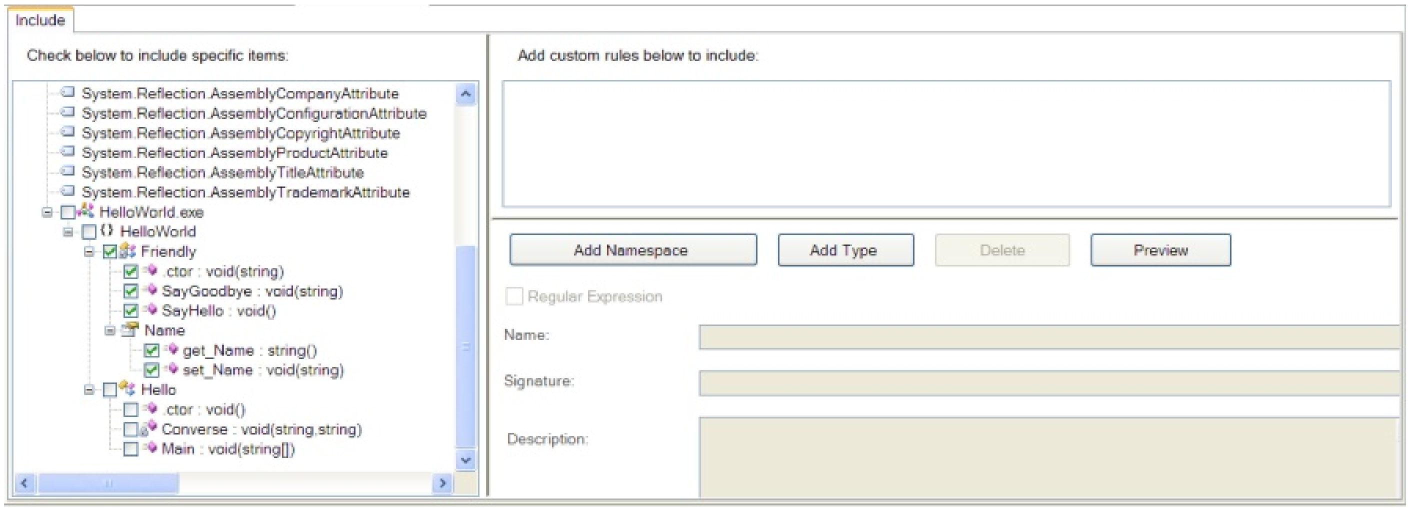
2.5.5.1 The String Encryption Include Tab

The *String Encryption Include* tab provides complete granular control over all parts of your program that you may wish to include in the string encryption process. From here, you can also disable string encryption altogether.

You may include specific items for string encryption by browsing the tree view of your application and checking the items you want to include. In addition, you may visually create your own custom rules for selecting multiple items for inclusion.

To help you fine-tune your inclusion rules, you can preview their effects at any time. The application tree view will shade all items selected for inclusion. You can preview the cumulative effects of all rules, or preview a specific rule that you select.

See the section on The Rules Editing Interface for detailed information about working with Inclusion and Exclusion Rules.


Include
Check below to include specific items:
System.Reflection.AssemblyCompanyAttribute
System.Reflection.AssemblyConfigurationAttribute
System.Reflection.AssemblyCopyrightAttribute
System.Reflection.AssemblyProductAttribute
System.Reflection.AssemblyTitleAttribute
System.Reflection.AssemblyTrademarkAttribute
HelloWorld.exe
HelloWorld
Friendly
.ctor : void(string)
SayGoodbye : void(string)
SayHello : void()
Name
get_Name : string()
set_Name : void(string)
Hello
.ctor : void()
Converse : void(string,string)
Main : void(string[])
Add custom rules below to include:
Add Namespace
Add Type
Delete
Preview
Regular Expression
Name:
Signature:
Description:

2.5.6 The Removal Editor

The *Removal* editor displays three configuration tabs:

- The *Include Triggers* tab, which is used to tell Dotfuscator about the entry points for your application, so it can determine which code can be safely removed.
- The *Conditional Includes* tab, which is used to tell Dotfuscator about types that should not be wholly removed, because of special cases such as reflection.
- The *Removal Options* tab, which is used to configure the removal report.

For new projects, the default setting for the Removal transform is **Disabled**.

2.5.6.1 Understanding Include Triggers and Conditional Includes

Dotfuscator can statically analyze an application to determine which elements are not actually used and remove those elements from the output binaries, reducing application size.

The static analysis works by traversing your code, starting at a set of methods called “triggers," or entry points. In general, any method that you expect external applications to call must be defined as a trigger. For example, in a simple standalone application, the **`Main`** method would be defined as a trigger. An assembly can have more than one trigger defined for it.

Note that turning on library mode for an assembly causes Dotfuscator to treat all visible types and members as entry points, automatically.

As Dotfuscator traverses each trigger method’s code, it notes which fields, methods, and types are being used. It then analyses all the called methods in a similar manner. The process continues until all called methods have been analyzed. Upon completion, Dotfuscator is able to determine a minimum set of types and their members necessary for the application to run. Only these types are included in the output assembly.

If Dotfuscator is unable to tell that certain methods are being called (due to reflection / XAML / etc.), then it may try to remove things that are required at runtime. To avoid this, you configure Include Triggers to tell Dotfuscator which class members (method, property, field, event) should be treated as "entry points" for the static analysis. Dotfuscator will preserve those members and all descendants of those members in the call graph.

Sometimes, though, this isn't the most optimal behavior. Consider an application that loads a set of types via reflection, casts them to an interface, and then invokes methods on that interface - a plugin model, essentially. Dotfuscator's static analysis won't identify the types that could potentially be loaded, but it does know which methods on that interface are going to be called.

In such a case, you should configure the types as Conditional Includes. Dotfuscator will include them and figure out that they implement the interface. If it determines that some of the methods in the interface are unused, it will prune those methods from the interface, and from all the implementations in the conditionally included types. Methods that weren't pruned will then be further analyzed for pruning, as usual.

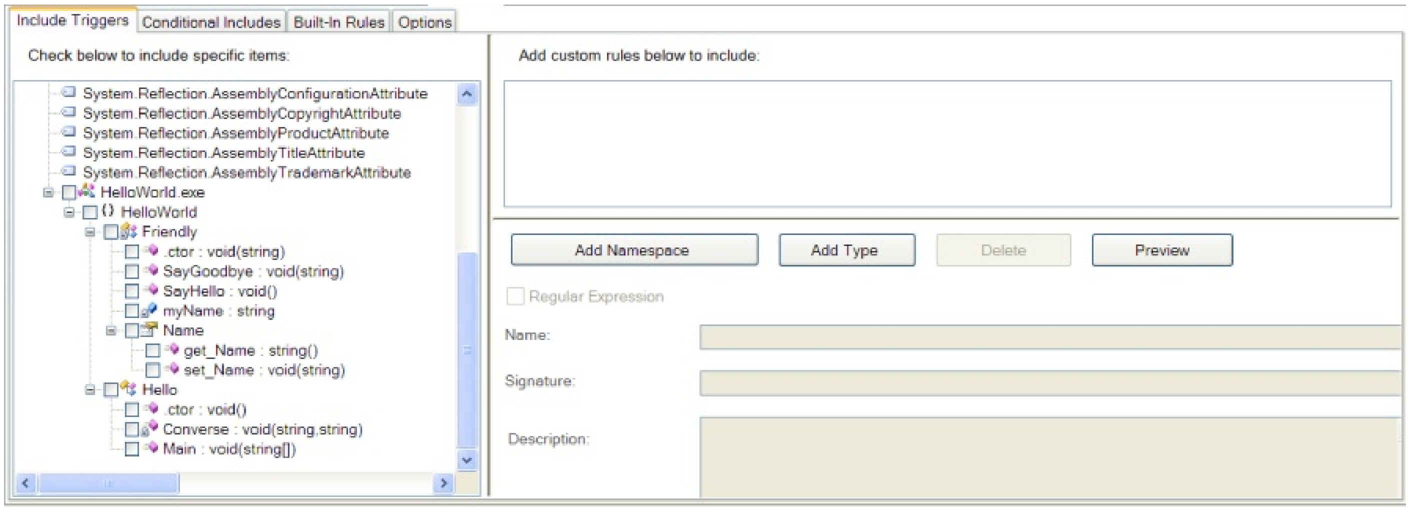
2.5.6.2 The Include Triggers Tab

The *Include Triggers* tab allows you to graphically specify all the methods that are to be used as application entry points ("triggers") for the pruning process.

You may include specific methods as entry points by browsing the tree view of your application and checking the items you want to include. In addition, you may visually create your own custom rules for selecting multiple methods for inclusion.

To help you fine-tune your inclusion rules, you can preview their effects at any time. The application tree view shades all items selected for inclusion. You can preview the cumulative effects of all rules, or preview a specific rule that you select.

See the section on The Rules Editing Interface for detailed information about working with Inclusion and Exclusion Rules.


Include Triggers
Conditional Includes
Built-In Rules
Options
Check below to include specific items:
System.Reflection.AssemblyConfigurationAttribute
System.Reflection.AssemblyCopyrightAttribute
System.Reflection.AssemblyProductAttribute
System.Reflection.AssemblyTitleAttribute
System.Reflection.AssemblyTrademarkAttribute
HelloWorld.exe
HelloWorld
Friendly
.ctor : void(string)
SayGoodbye : void(string)
SayHello : void()
myName : string
Name
get_Name : string()
set_Name : void(string)
Hello
.ctor : void()
Converse : void(string,string)
Main : void(string[])
Add custom rules below to include:
Add Namespace
Add Type
Delete
Preview
Regular Expression
Name:
Signature:
Description:

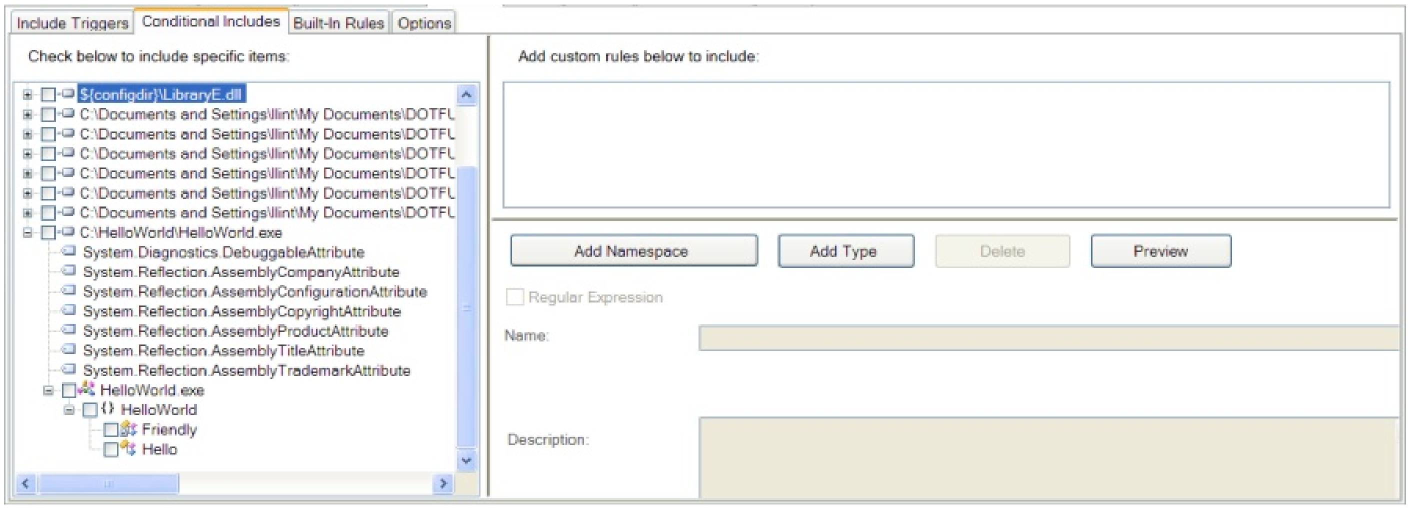
2.5.6.3 The Conditional Includes Tab

The *Conditional Includes* tab allows you to graphically specify all the types that should be conditionally included in the pruning process. Please see Understanding Include Triggers and Conditional Includes for a deeper explanation of this feature.

You may conditionally include specific types by browsing the tree view of your application and checking the items you want to include. In addition, you may visually create your own custom rules for selecting multiple types for inclusion.

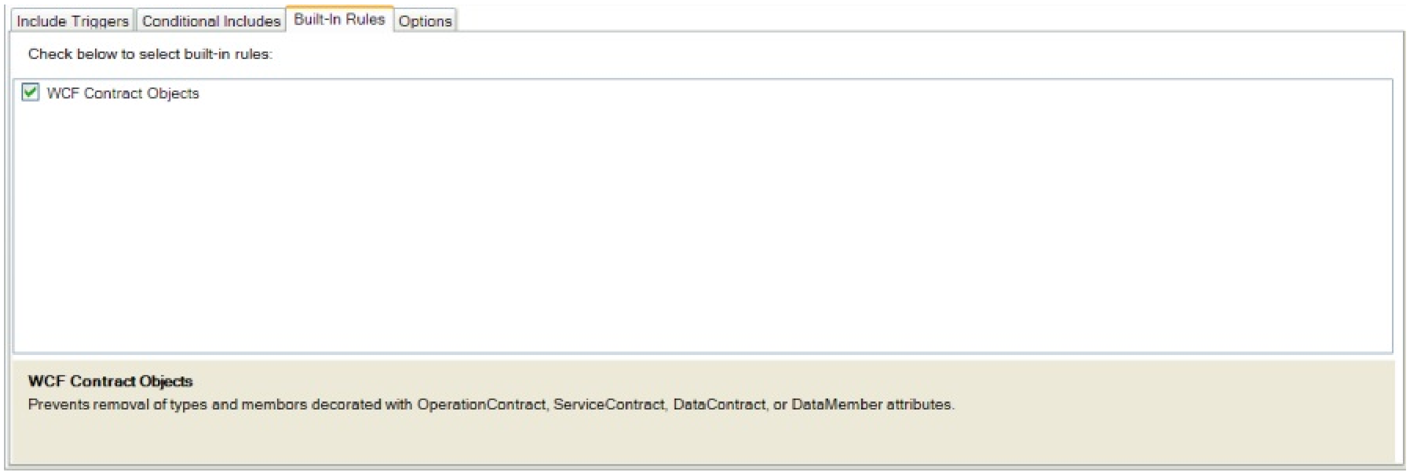
To help you fine-tune your inclusion rules, you can preview their effects at any time. The application tree view shades all items selected for inclusion. You can preview the cumulative effects of all rules, or preview a specific rule that you select.

See the section on The Rules Editing Interface for detailed information about working with Inclusion and Exclusion Rules.


Include Triggers
Conditional Includes
Built-In Rules
Options
Check below to include specific items:
${configdir}\LibraryE.dll
C:\Documents and Settings\llint\My Documents\DOTFU
C:\Documents and Settings\llint\My Documents\DOTFU
C:\Documents and Settings\llint\My Documents\DOTFU
C:\Documents and Settings\llint\My Documents\DOTFU
C:\Documents and Settings\llint\My Documents\DOTFU
C:\Documents and Settings\llint\My Documents\DOTFU
C:\HelloWorld\HelloWorld.exe
System.Diagnostics.DebuggableAttribute
System.Reflection.AssemblyCompanyAttribute
System.Reflection.AssemblyConfigurationAttribute
System.Reflection.AssemblyCopyrightAttribute
System.Reflection.AssemblyProductAttribute
System.Reflection.AssemblyTitleAttribute
System.Reflection.AssemblyTrademarkAttribute
HelloWorld.exe
HelloWorld
Friendly
Hello
Add custom rules below to include:
Add Namespace
Add Type
Delete
Preview
Regular Expression
Name:
Description:

2.5.6.4 The Built-In Rules Tab

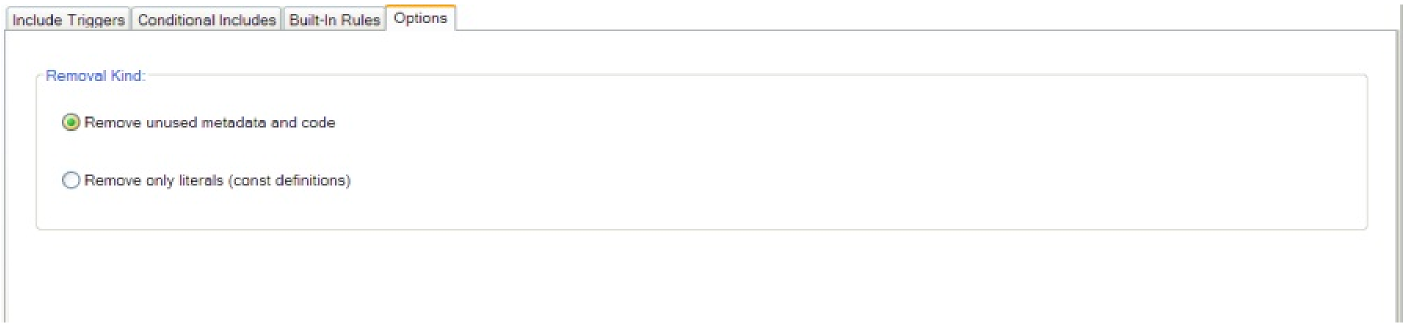
Dotfuscator’s *Built-In Rules* removal tab displays include trigger rules and conditional inclusion rules defined in **%ProgramData%\PreEmptive Solutions\Common\dotfuscatorReferenceRule_v1.4.xml**. These standard rules apply to specific application types or technologies. Each rule has a description that displays on the form when the rule is selected. You can apply a built-in rule to your project by checking its checkbox.



WCF Contract Objects
Prevents removal of types and members decorated with OperationContract, ServiceContract, DataContract, or DataMember attributes.

2.5.6.5 The Options Tab

The *Removal Options* tab is used to select the kind of removal you want to occur.



Removal Kind:

[✓ Remove unused metadata and code]

Remove only literals (const definitions)

Selecting *Remove only literals (const definitions)* will perform pruning only on constant declarations. Selecting *Remove unused metadata and code* follows the usual algorithm for determining unused methods and fields and removing them, as well as pruning constant definitions.

2.5.6.5.1 Constant-Only Pruning

You may encounter situations where you may not wish to configure pruning on an assembly, but still wish to achieve some of the attack surface and assembly size reduction goals of pruning. In these cases, constant-only pruning is an ideal compromise. During constant-only pruning, Dotfuscator will only prune constant declarations (`const` fields) from the input assemblies. Unused types, methods, and fields will not be discovered, and will be propagated to the output assembly.

Constant-only pruning is safe to do in many situations where full pruning is not desired. During compilation, .NET compilers will replace references to `const` fields in code with the actual values of those fields. The constant declarations remain in the assembly only to support being referenced by external assemblies or being accessed via reflection. If you do not need to support these scenarios, it is generally safe to enable constant-only pruning.

2.5.6.6 Removal Report

Dotfuscator generates a removal report in XML format that lists all input assemblies and how each was pruned. Each assembly listing has a listing of types and their members (methods, fields, properties, etc.) along with an attribute indicating whether the item was pruned or not. The report also describes how managed resources attached to each assembly were pruned. At the end, the report provides a statistics section regarding the overall effectiveness of pruning.

The removal report is most useful when converted to HTML, using the default transform (or one of your own). The default transform produces a browsable, cross referenced report that indicates pruned items in red.

The elements of the removal report are similar to those in the map file. A few things are noteworthy:

- The report includes pruning status of: types, methods, fields, properties, and managed resources.
- If a type was pruned, then obviously all its members (methods, fields and properties) were pruned.
- In type names, nested class names are separated from the parent using the "`/`" character.
- Constructors are named `.ctor`, while static constructors (a.k.a. static initializers, class constructors, etc) are named `.cctor`.

2.5.7 The Linking Editor

The Linking editor provides easy drag and drop functionality that allows you to quickly map your input assemblies to one or more output assemblies. You can then configure the linker for each output assembly.

> For new projects, the default setting for the Linking transform is **Disabled**.

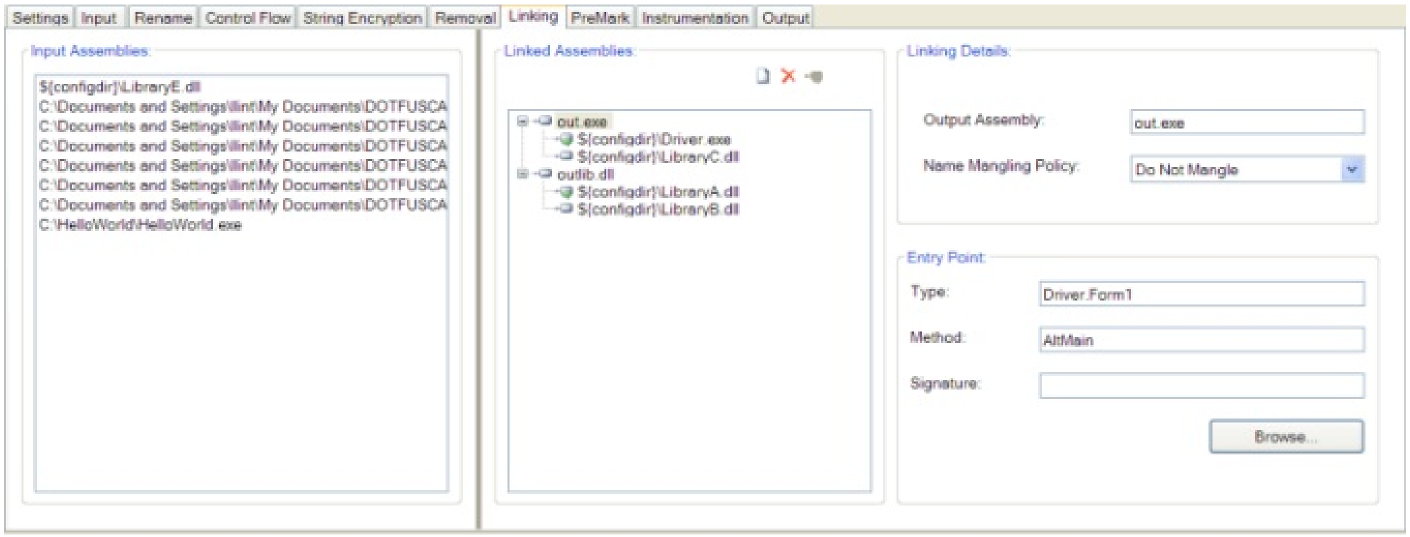
2.5.7.1 Input Assemblies and Linked Assemblies

To create and configure an output assembly for linking, follow these steps.

- Click the **Create New Assembly** button on the *Linked Assemblies* toolbar. An entry for your new assembly appears on the linked assemblies list.
- Give your new output assembly a name by typing it in the *Output Assembly* field in the *Linking Details* section. You do not have to type path information; like all output assemblies, it will be written to the destination directory.
- Select the assemblies you want to link from the *Input Assemblies* list and drag them to the new assembly in the *Linked Assemblies* list. They will disappear from the *Input Assemblies* list and appear as child nodes of the new assembly.
- Right click on the input assembly that you wish to mark as your prime assembly and select **Set Primary Assembly** from the context menu. You can also do this by selecting the prime assembly and using the toolbar button. Prime assemblies are indicated with a tag icon.
- If needed, set the Name Mangling Policy and Entry Points for the new assembly.
- Repeat the process for each output assembly you want to create.

You can remove an assembly from the *Linked Assemblies* list by selecting it and using the delete button on the toolbar or by pressing the **Delete** key.

The screenshot below shows an example with multiple linked outputs. First, input assemblies **Driver.exe** and **LibraryC.dll** are linked into **out.exe**; next, **LibraryA.dll** and **LibraryB.dll** are linked into **outlib.dll**; and last, **LibraryE.dll** "passes through" without being linked.



Input Assemblies:

```
${configdir}\LibraryE.dll
C:\Documents and Settings\lint\My Documents\DOTFUSCA
C:\Documents and Settings\lint\My Documents\DOTFUSCA
C:\Documents and Settings\lint\My Documents\DOTFUSCA
C:\Documents and Settings\lint\My Documents\DOTFUSCA
C:\Documents and Settings\lint\My Documents\DOTFUSCA
C:\Documents and Settings\lint\My Documents\DOTFUSCA
C:\HelloWorld\HelloWorld.exe
```

Linked Assemblies:

```
out.exe
    ${configdir}\Driver.exe
    ${configdir}\LibraryC.dll
outlib.dll
    ${configdir}\LibraryA.dll
    ${configdir}\LibraryB.dll
```

Linking Details:

Output Assembly: out.exe

Name Mangling Policy: Do Not Mangle

Entry Point:

Type: Driver.Form1

Method: AltMain

Signature:

2.5.7.1.1 Prime Assemblies

When you set up linking, you must specify one of the input assemblies as the **prime assembly**. Dotfuscator applies the manifest information (e.g. version number, public key, etc.) contained in the prime assembly to the newly created output assembly.

2.5.7.1.2 Name Mangling

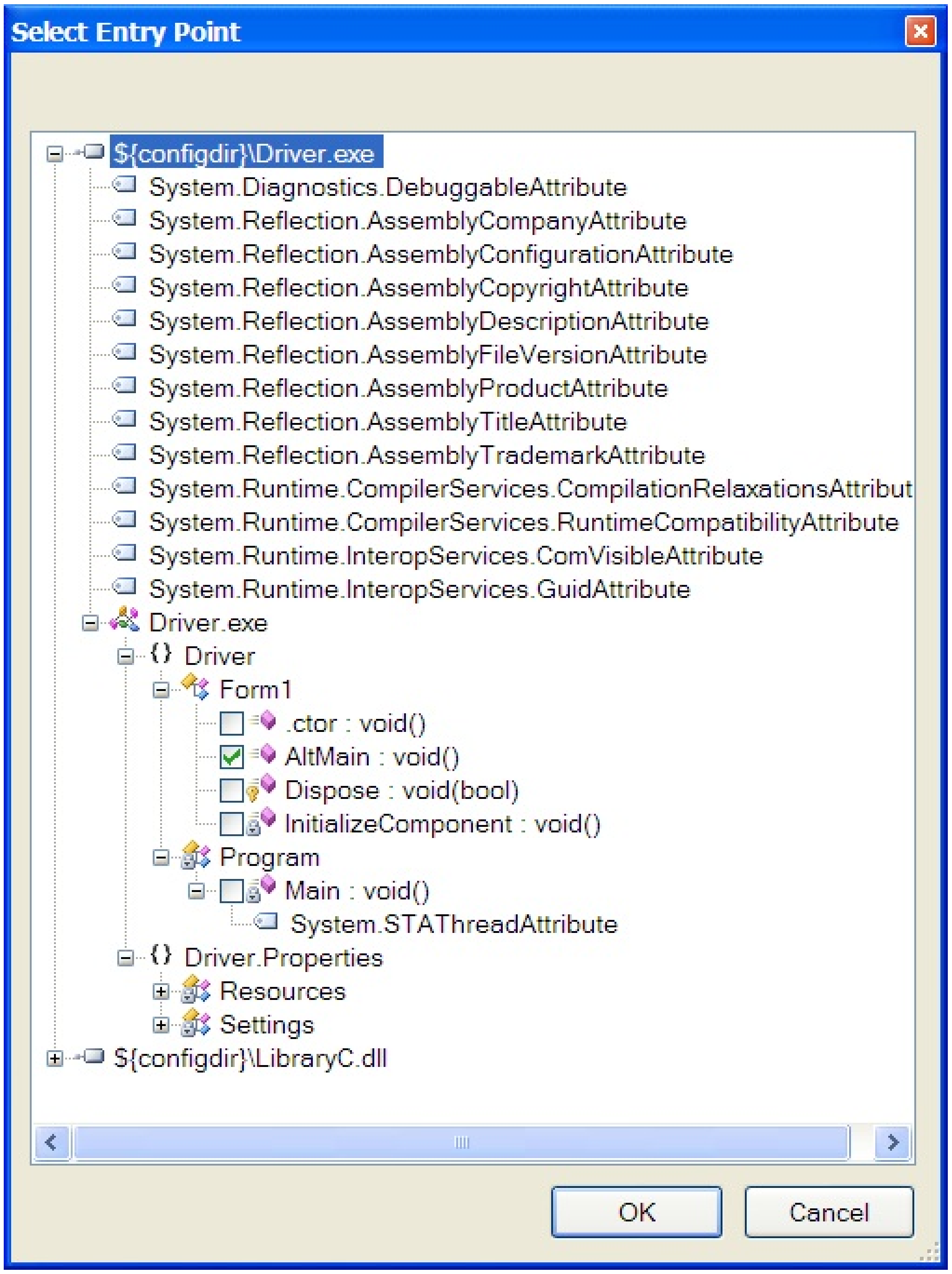
When the linker is merging assemblies, the linker sometimes encounters situations where a name needs to be changed in order to prevent a naming collision. For example, if two of the input assemblies contain private classes with identical names then the linker must change one of the names in order to merge the assemblies.

In most cases, Dotfuscator can safely mangle names (when at least one of the original names is not visible outside its assembly); however, if Dotfuscator sees a case in which there are two visible types with the same name, it cannot safely mangle either name without guidance. It is only in this case where the name mangling policy is used. The default is to halt the build with an error message. Other options are to mangle one of the names and issue a warning, or silently mangle the names.

2.5.7.2 Setting Entry Points

If you need to set an entry point for your linked output assembly, you can use the *Select Entry Point* dialog by pressing the **Browse** button in the linking editor's *Entry Point* section.

This dialog shows a graphical view of the all the methods that will be available in the linked output assembly. Checking a method selects it as an entry point. The dialog will fill in the rest of the information in the *Entry Point* section.



```
${configdir}\Driver.exe
    System.Diagnostics.DebuggableAttribute
    System.Reflection.AssemblyCompanyAttribute
    System.Reflection.AssemblyConfigurationAttribute
    System.Reflection.AssemblyCopyrightAttribute
    System.Reflection.AssemblyDescriptionAttribute
    System.Reflection.AssemblyFileVersionAttribute
    System.Reflection.AssemblyProductAttribute
    System.Reflection.AssemblyTitleAttribute
    System.Reflection.AssemblyTrademarkAttribute
    System.Runtime.CompilerServices.CompilationRelaxationsAttribut
    System.Runtime.CompilerServices.RuntimeCompatibilityAttribute
    System.Runtime.InteropServices.ComVisibleAttribute
    System.Runtime.InteropServices.GuidAttribute
    Driver.exe
        Driver
            Form1
                [ ] .ctor : void()
                [✓] AltMain : void()
                [ ] Dispose : void(bool)
                [ ] InitializeComponent : void()
            Program
                [ ] Main : void()
                    System.STAThreadAttribute
        Driver.Properties
            Resources
            Settings
${configdir}\LibraryC.dll
```

2.5.7.2.1 Entry Points

In .NET, an executable assembly must have a method marked in the metadata as the entry point (typically this method is called `Main`, but it can have any name.). This is the method that the CLR calls when the assembly is run. In some cases you need to specify an entry point for each linked output assembly. The table below summarizes the rules for entry points.

Inputs	Output	Entry Point Rule
All are EXEs	**Is an EXE**	A user specified entry point is required. The linker will remove all entry points on input assemblies and apply the user specified entry point to the output assembly.
All are DLLs	**Is an EXE**	A user specified entry point is required.
Mixed EXEs and DLLs	**Is an EXE**	A user specified entry point is required, except if there is only one input exe; in which case the linker will use its entry point. A user specified entry point will override the default.
Anything	**Is a DLL**	A user specified entry point is not used. The linker will remove all entry points on inputs.

2.5.8 The PreMark Editor

The *PreMark* editor allows you to select assemblies for watermarking and to set up the watermark that will be applied to the selected assemblies.

For new projects, the default setting for the PreMark transform is **Disabled**.

Selecting Assemblies

The list of input assemblies is displayed on the left. Checking the box in front of an assembly selects it for watermarking.

Watermarking Options

You can choose to encrypt your watermark string before it is applied to your input assemblies. The encryption is based on a pass phrase that you provide. When you check the **Encrypt Using Passphrase** box, the passphrase entry text box becomes enabled and you can then enter your passphrase.

Note: Encryption can increase the size of the watermark that is applied to your input assemblies.

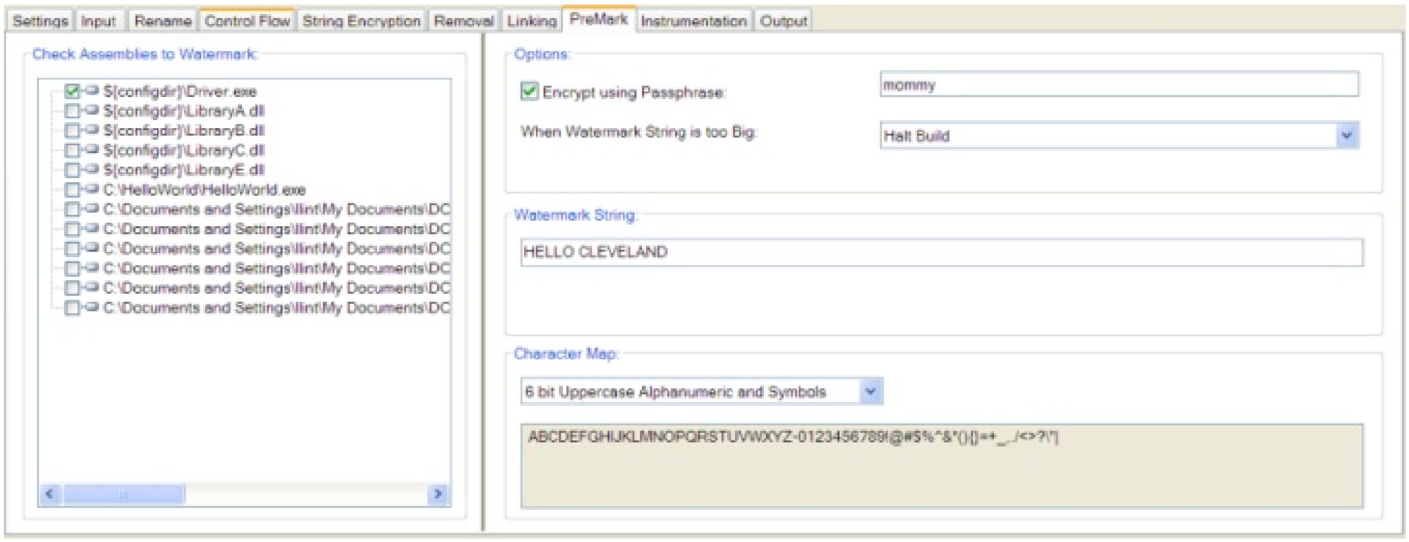
Dotfuscator cannot predict whether an encoded and potentially encrypted watermark string will fit in a given output assembly. At watermarking time, if the string does not fit, you can tell Dotfuscator what to do via the **When Watermark String is Too Big** dropdown. You can select whether it should automatically truncate the watermark string and issue a warning, or halt the build with an error.

Character Map

Your watermark string must be encoded to bytes before it is applied to an assembly. Dotfuscator provides several built in Character Maps that are optimized for small, commonly used, character sets. Using one of them will allow larger watermark strings to fit in any given assembly. If none of the specialized character maps fit your needs, you can select the standard UTF-8 encoding.

Watermark String

When entering your watermark string, the user interface will notify you if you type a character that is not included in the currently selected character map.



Check Assemblies to Watermark:

- [✓] $[configdir]\Driver.exe
- [ ] $[configdir]\LibraryA.dll
- [ ] $[configdir]\LibraryB.dll
- [ ] $[configdir]\LibraryC.dll
- [ ] $[configdir]\LibraryE.dll
- [ ] C:\HelloWorld\HelloWorld.exe
- [ ] C:\Documents and Settings\lint\My Documents\DC
- [ ] C:\Documents and Settings\lint\My Documents\DC
- [ ] C:\Documents and Settings\lint\My Documents\DC
- [ ] C:\Documents and Settings\lint\My Documents\DC
- [ ] C:\Documents and Settings\lint\My Documents\DC
- [ ] C:\Documents and Settings\lint\My Documents\DC

Options:

[✓] Encrypt using Passphrase: mommy

When Watermark String is too Big: Halt Build

Watermark String: HELLO CLEVELAND

Character Map: 6 bit Uppercase Alphanumeric and Symbols

ABCDEFGHIJKLMNOPQRSTUVWXYZ-0123456789!@#$%^&*(){}=+_./<>?\"|

2.5.8.1 Watermark String Length

The maximum size of the watermark string is governed by your configuration options and by the complexity of the target assembly. In general, bigger strings fit in bigger assemblies. Dotfuscator uses character encodings, called character maps, to minimize the number of bits required to encode a character; a small character encoding allows you to create a longer watermark string for a given assembly.

In addition, the encryption algorithm has a fixed block size. If you choose to encrypt the watermark string, the maximum length of your watermark string may be smaller than it is without encryption.

It is not possible for Dotfuscator to predict the maximum watermark string length until the output assemblies have been generated. You can tell Dotfuscator what to do during a build when your watermark string will not fit in the output assembly. The default setting truncates the string so it fits and prints a warning message to the output window. You can also tell Dotfuscator to stop the build with an error. In both cases, the message indicates the maximum watermark size.

2.5.8.2 Character Maps

Dotfuscator defines several character maps you can use to encode your watermark string. If space is an issue, you can choose a smaller encoding at the expense of the number of different characters you can use in your string.

Name	Description	Bits/Character
`4bit-a`	4 bit Hexadecimal	4
`6bit-a`	6 bit Uppercase Alphanumeric and symbols	6
`6bit-b`	6 bit Alphanumeric	6
`7bit-a`	7 bit Alphanumeric and symbols	7
`UTF-8`	Any character	variable

Dotfuscator’s user interface displays the specific characters defined in each character map.

2.5.8.3 Extracting a Watermark

Dotfuscator ships with a command line tool called **`premark`** that accepts an assembly as input and outputs the watermark if any. The tool is installed into the same directory as Dotfuscator

Extracting a Watermark

```
premark [options] assembly1[,assembly2, ...]

assembly1,...:  A list of .NET assemblies or modules.

Options:
/a              Ask for passphrase.
/p=passphrase   Passphrase to use when decrypting watermark string
                (none for no encryption).
```

In addition, there is an MSBuild task defined for extracting a watermark. See PreMark Task.

2.5.9 The Rules Editing Interface

The Dotfuscator Graphical User Interface uses a common interface to graphically specify rules for including and excluding elements in your application. The rules editing interface is used to set rules for the following operations:

- Renaming exclusion rules
- Control Flow exclusion rules
- String Encryption inclusion rules

- Removal Trigger Method selection rules
- Removal Conditional Includes selection rules

This section explains how to get the most out of the rules editing interface.

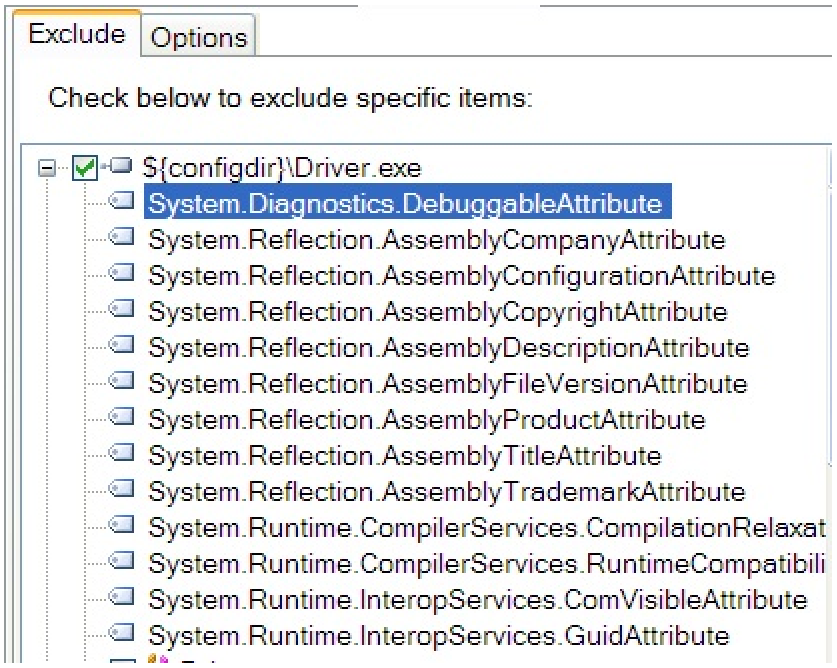
There are two methods of creating rules. The first is by checking individual elements in the application tree view. By doing this, you generate a simple rule that selects that particular element. The second method is by adding “nodes” to the rule editing view. This type of rule is more powerful and customizable. You can use regular expressions and other selection criteria, based on the type of rule. This method also provides ways to preview the items selected by each rule.

2.5.9.1 Selecting Individual Elements

You can create selection rules for individual elements by simply checking the box next to the item in the application tree view. You can check assemblies, modules, types, methods, and fields in this manner.

Assemblies

The top-most nodes in the application tree view represent the packages or assemblies. If you check an assembly node, all child nodes become checked. This reflects the fact that selecting an assembly means that you are selecting all items contained in that assembly: modules, types, and their members.

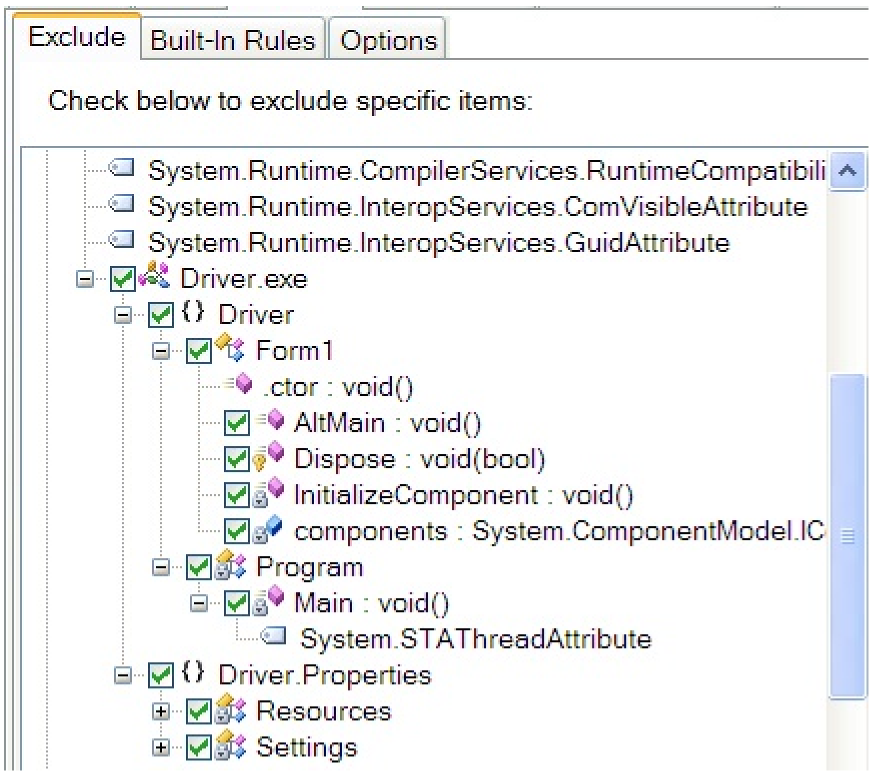


Check below to exclude specific items:

```
${configdir}\Driver.exe
    System.Diagnostics.DebuggableAttribute
    System.Reflection.AssemblyCompanyAttribute
    System.Reflection.AssemblyConfigurationAttribute
    System.Reflection.AssemblyCopyrightAttribute
    System.Reflection.AssemblyDescriptionAttribute
    System.Reflection.AssemblyFileVersionAttribute
    System.Reflection.AssemblyProductAttribute
    System.Reflection.AssemblyTitleAttribute
    System.Reflection.AssemblyTrademarkAttribute
    System.Runtime.CompilerServices.CompilationRelaxat
    System.Runtime.CompilerServices.RuntimeCompatibili
    System.Runtime.InteropServices.ComVisibleAttribute
    System.Runtime.InteropServices.GuidAttribute
```

Modules

The nodes immediately below an assembly node in the application tree view represent the modules that make up the assembly (in most cases there is one module per assembly). If you check a module node, all child nodes become checked. This reflects the fact that selecting a module means selecting all items contained in that module: “global” methods and fields, types, and their members.

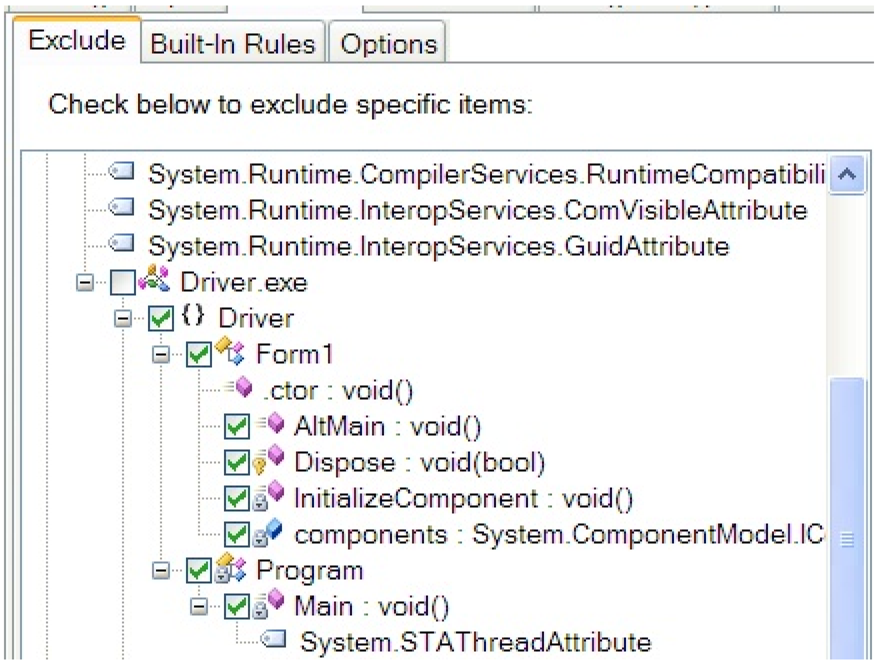


Check below to exclude specific items:

```
System.Runtime.CompilerServices.RuntimeCompatibili
System.Runtime.InteropServices.ComVisibleAttribute
System.Runtime.InteropServices.GuidAttribute
Driver.exe
  Driver
    Form1
      .ctor : void()
      AltMain : void()
      Dispose : void(bool)
      InitializeComponent : void()
      components : System.ComponentModel.IC
    Program
      Main : void()
        System.STAThreadAttribute
  Driver.Properties
    Resources
    Settings
```

Namespaces

Namespace nodes are child nodes of module nodes in the application tree view. If you check a namespace node, all child nodes become checked. This reflects the fact that selecting a namespace means selecting all items contained in that namespace: types and their members.



Check below to exclude specific items:

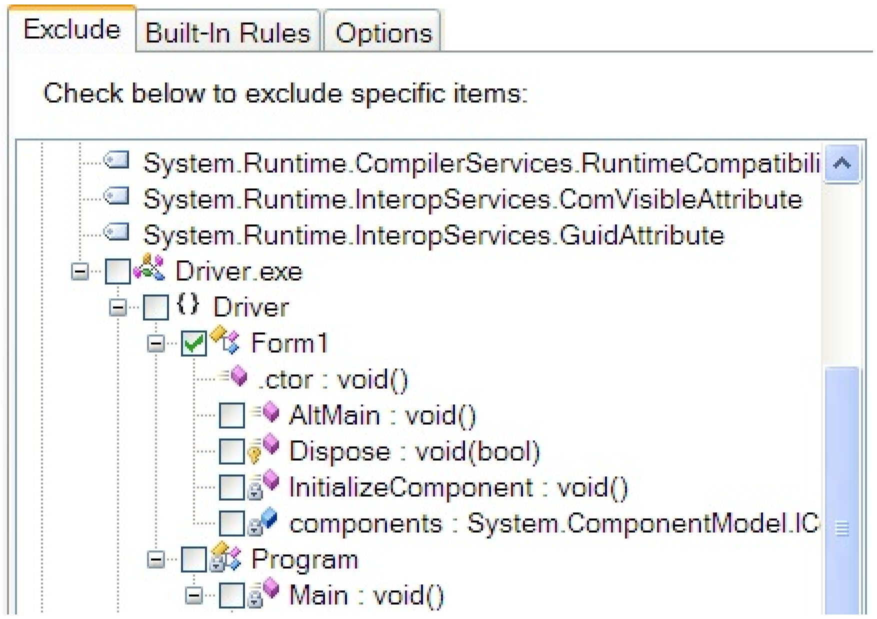
```
System.Runtime.CompilerServices.RuntimeCompatibili
System.Runtime.InteropServices.ComVisibleAttribute
System.Runtime.InteropServices.GuidAttribute
[ ] Driver.exe
  [✓] Driver
    [✓] Form1
        .ctor : void()
      [✓] AltMain : void()
      [✓] Dispose : void(bool)
      [✓] InitializeComponent : void()
      [✓] components : System.ComponentModel.IC
    [✓] Program
      [✓] Main : void()
          System.STAThreadAttribute
```

Types

Type nodes appear under module or namespace nodes. Nested types are represented at this same level, with a name prefixed with the parent type name(s) delimited with the '/' character. If you check a type node, one of two things happens, depending on what type of rule you are creating.

If you are creating a renaming exclusion rule, child nodes remain unchecked. This reflects the fact that types are selected independently of their members for renaming exclusion rules. Checking a type node will generate a rule that excludes *just the type name* from renaming.

If you are specifying any other kind of rule, all child nodes will also become checked. This reflects the fact that in these cases, selecting a type means that you are in fact selecting all members defined by that type.



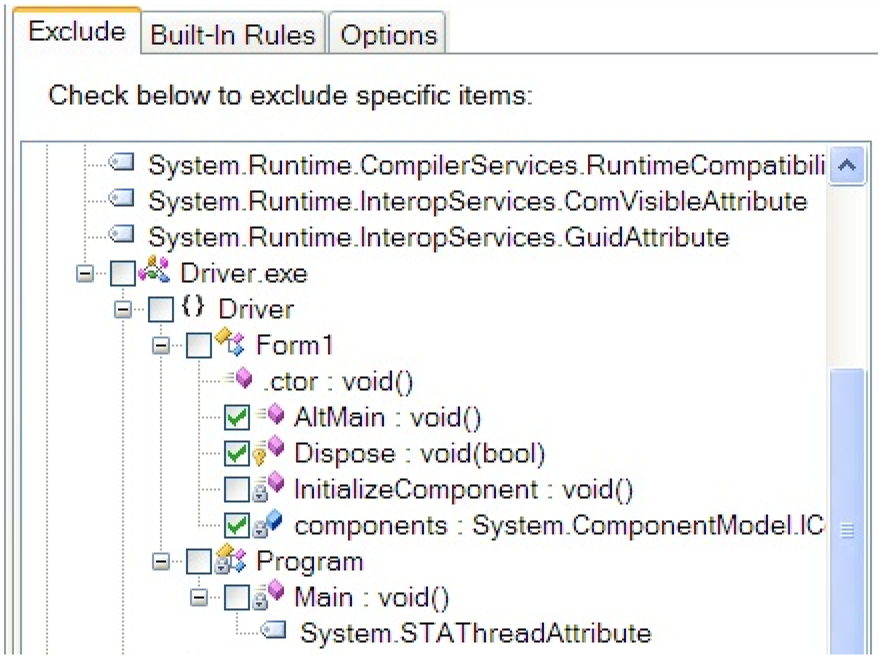
Exclude | Built-In Rules | Options

Check below to exclude specific items:

- System.Runtime.CompilerServices.RuntimeCompatibili
- System.Runtime.InteropServices.ComVisibleAttribute
- System.Runtime.InteropServices.GuidAttribute
- [ ] Driver.exe
  - [ ] Driver
    - [✓] Form1
      - .ctor : void()
      - [ ] AltMain : void()
      - [ ] Dispose : void(bool)
      - [ ] InitializeComponent : void()
      - [ ] components : System.ComponentModel.IC
    - [ ] Program
      - [ ] Main : void()

Members

Members can be methods, fields, properties, or events. Member nodes can appear under module nodes in the case of "global" members; more commonly, they appear under type nodes. Checking a member node will generate a rule that selects that member.



Exclude | Built-In Rules | Options

Check below to exclude specific items:

- System.Runtime.CompilerServices.RuntimeCompatibili
- System.Runtime.InteropServices.ComVisibleAttribute
- System.Runtime.InteropServices.GuidAttribute
- [ ] Driver.exe
  - [ ] Driver
    - [ ] Form1
      - .ctor : void()
      - [✓] AltMain : void()
      - [✓] Dispose : void(bool)
      - [ ] InitializeComponent : void()
      - [✓] components : System.ComponentModel.IC
    - [ ] Program
      - [ ] Main : void()
        - System.STAThreadAttribute

2.5.9.2 Creating Custom Rules

You can create custom rules by adding nodes to the rule editing view. Depending on the type of rule, you can attach regular expressions and/or other selection criteria to the rule. Once the rule is configured, you can preview its effects by right clicking on the node and selecting **Preview** from the menu. Items selected by the rule will be shaded in the application tree view. Each type of rule is covered in the following sections.

2.5.9.2.1 Selecting By Namespace

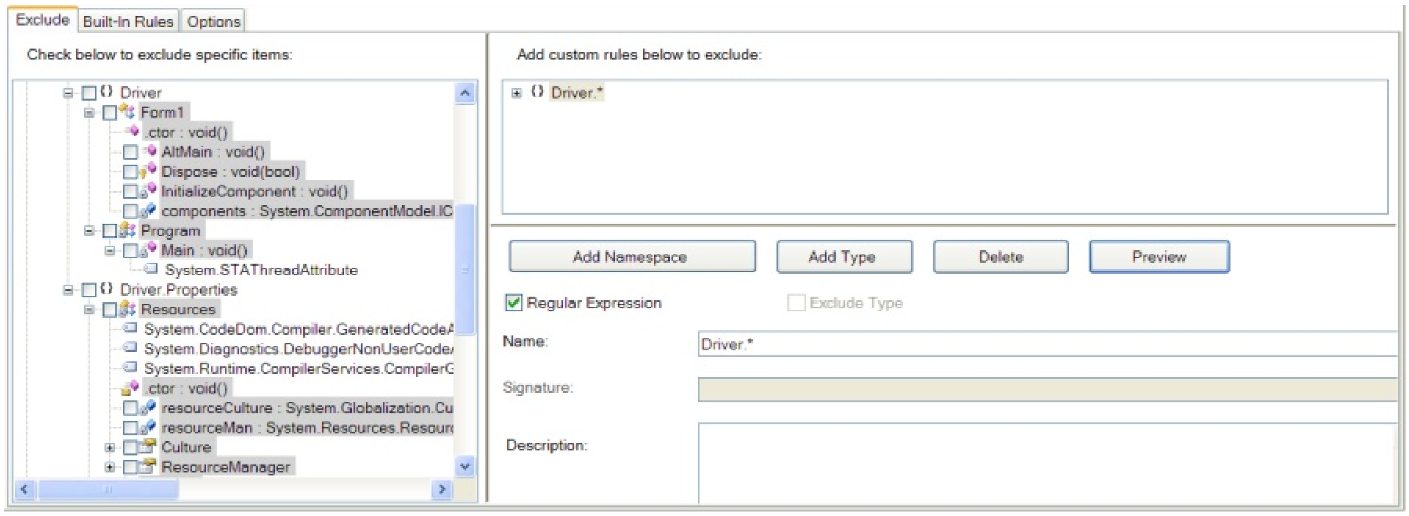
A namespace rule will select all types and their members in matching namespaces.

Namespace Name

You create a namespace rule by clicking the **Add Namespace** button, then typing a name in the *Name* field. The name will be interpreted as a regular expression if the **Regular Expression** checkbox is checked; otherwise the name will be interpreted literally (and thus will match at most one namespace).

Namespace Rule Node

The corresponding node displayed in the rule editing view has a child element that indicates whether the rule is a regular expression. You can preview the items selected by the rule by right clicking on the node and selecting the preview option from the menu.


Exclude
Built-In Rules
Options
Check below to exclude specific items:
Driver
Form1
.ctor : void()
AltMain : void()
Dispose : void(bool)
InitializeComponent : void()
components : System.ComponentModel.IC
Program
Main : void()
System.STAThreadAttribute
Driver.Properties
Resources
System.CodeDom.Compiler.GeneratedCodeA
System.Diagnostics.DebuggerNonUserCode
System.Runtime.CompilerServices.CompilerG
.ctor : void()
resourceCulture : System.Globalization.Cu
resourceMan : System.Resources.Resour
Culture
ResourceManager
Add custom rules below to exclude:
Driver.*
Add Namespace
Add Type
Delete
Preview
Exclude Type
Signature:
Description:

☑ Regular Expression

Name: Driver.*

2.5.9.2.2 Selecting By Type

A type rule will select differently depending on what type of rule you are creating.

If you are creating a renaming exclusion rule, the rule will select just the type name for exclusion (provided the **ExcludeType** checkbox is checked), leaving members alone.

If you are specifying any other kind of rule, the rule will select zero or more types and all their members. This reflects the fact that in these cases, selecting a type means that you are in fact selecting all members defined by that type.

Type Name

You create a type rule by clicking the **Add Type** button, then typing a name in the *Name* field. The name will be interpreted as a regular expression if the **Regular Expression** checkbox is checked; otherwise the name will be interpreted literally. The name must be a fully qualified type name that includes the namespace and parent class information if it is a nested type.

Type Attribute Specifier

In addition to type name, you can also select based on type attribute specifiers, using the values provided in the "*Spec*" list box. A **'-'** preceding an attribute specifier negates the attribute (*i.e.* it selects all types that do not have the specified attribute). You can select multiple attributes from the list; the criteria implied by multiple selections are logically **`AND`**-ed together. For example, you can select types that are both public and abstract by selecting **+public** and **+abstract** from the list.

The attribute specifications are logically **`AND`**-ed with the type name, so if you want to select all types with a specific set of attributes, you need to provide a regular expression for the type name that selects all types (*i.e.* "**`.*`**").

Exclude Type Checkbox

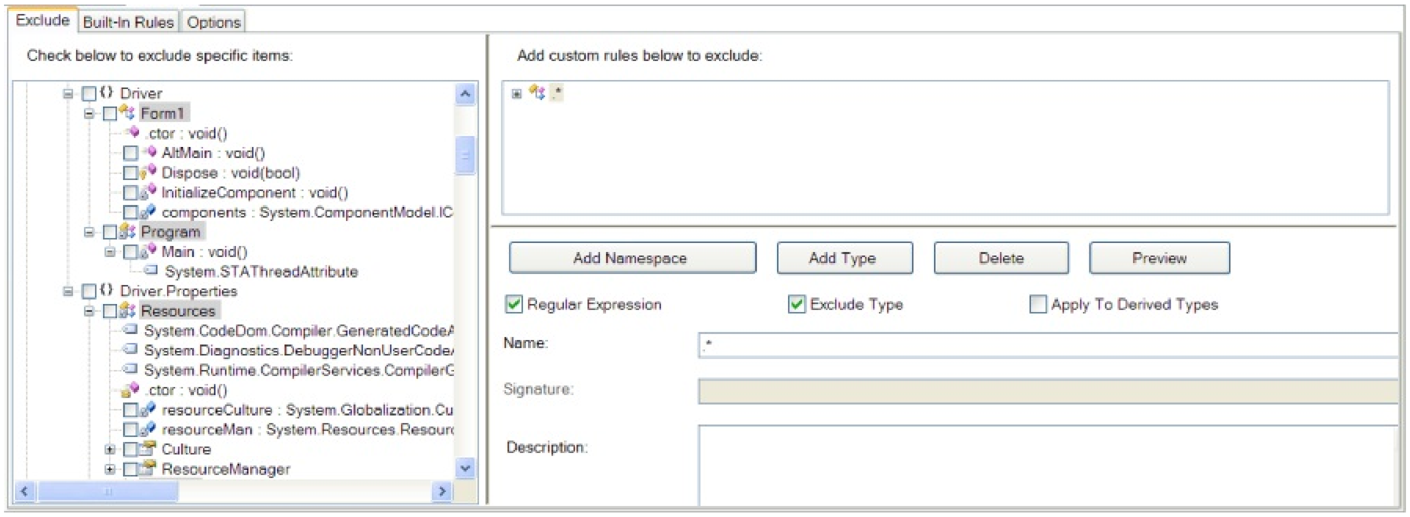
The **Exclude Type** checkbox is only active if you are working with renaming exclusion rules. If checked, the rule will exclude the names of matching types from renaming and allow you to provide additional rules for selecting members of matching types. If left unchecked, the rule will still select matching types for the purposes of applying rules to members of the types, but it will not select the type name. In this manner, you can write renaming exclusion rules that exclude methods and fields, but allow type names to be obfuscated.

Apply to Derived Types Checkbox

The **Apply to Derived Types** checkbox is only active if you are working with renaming or pruning rules. If checked, the rule will additionally exclude the child classes of matching types from renaming or pruning. In this manner, you can write renaming exclusion rules that exclude entire inheritance hierarchies.

Type Rule Node

The corresponding node displayed in the rule editing view has a child element that indicates whether the rule is a regular expression and whether the rule has attribute specifiers associated with it. You can preview the types selected by the rule by right clicking on the node and selecting the **Preview** option from the menu.


Exclude
Built-In Rules
Options
Check below to exclude specific items:
Driver
Form1
.ctor : void()
AltMain : void()
Dispose : void(bool)
InitializeComponent : void()
components : System.ComponentModel.IC
Program
Main : void()
System.STAThreadAttribute
Driver.Properties
Resources
System.CodeDom.Compiler.GeneratedCodeA
System.Diagnostics.DebuggerNonUserCodeA
System.Runtime.CompilerServices.CompilerG
.ctor : void()
resourceCulture : System.Globalization.Cu
resourceMan : System.Resources.Resour
Culture
ResourceManager
Add custom rules below to exclude:
.*
Add Namespace
Add Type
Delete
Preview
Regular Expression
Exclude Type
Apply To Derived Types
Name:
Signature:
Description:

Settings shown: Regular Expression [✓], Exclude Type [✓], Apply To Derived Types [ ]; Name: .*

In the screen shot, a type rule is defined that selects the names of all concrete (not abstract) types for exclusion from renaming.

2.5.9.2.3 Selecting By Method

Method rules are qualified by type rules, so they appear in the rules view as children of type nodes. A method rule will select all methods (in all types matched by the parent type rule) that match your criteria. Supported matching criteria include method name, method attributes, and signature.

Method Name

You create a method rule by right clicking on the parent type rule’s node and selecting **Add Method**, then typing a name in the *Name* field. The name will be interpreted as a regular expression if the **Regular Expression** checkbox is checked; otherwise the name will be interpreted literally.

Method Attribute Specifier

In addition to method name, you can also select based on method attribute specifiers, using the values provided in the *Attribute Specifier* list box. A ‘**-**‘ preceding an attribute specifier negates the attribute (*i.e.* it selects all methods that do not have the specified attribute). You can select multiple attributes from the list; the criteria implied by multiple selections are logically `AND`-ed together (that is, the set of selected methods is the intersection of all methods that match each attribute specifier.). For example, you can select methods that are both public and virtual by selecting **+public** and **+virtual** from the list.

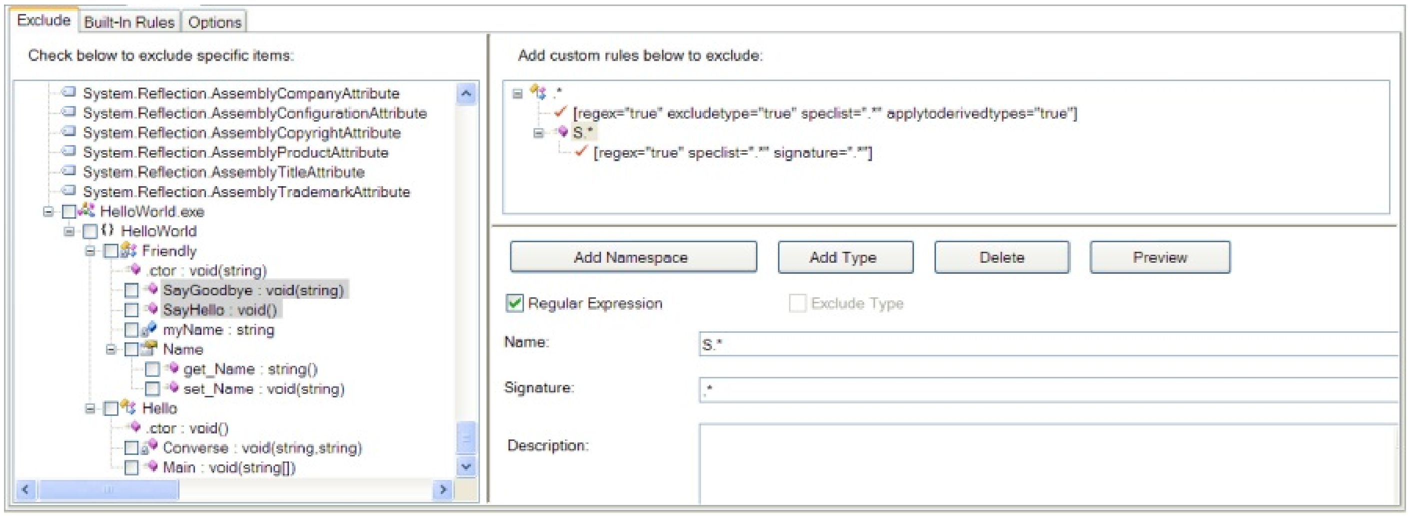
The attribute specifications are logically `AND`-ed with the method name and signature list, so if you want to select all methods with a specific set of attributes, you need to provide a regular expression for the method name that selects all methods (*i.e.* ".*").

Method Signature

You can also select methods by signature. A signature specifies both the return type and the parameter types of the method. The method signature reduces the scope of the method rule, so if you want to create a rule that selects methods regardless of signature, you need to provide a regular expression for the signature that selects all signatures (*i.e.* ".*****"). This is the default value.

Method Rule Node

The corresponding method node displayed in the rule editing view has a child element that indicates whether the rule is a regular expression and whether the rule has attribute specifiers, and/or a signature associated with it. You can preview the items selected by the rule by right clicking on the node and selecting the **Preview** option from the menu.


Exclude
Built-In Rules
Options
Check below to exclude specific items:
Add custom rules below to exclude:
Add Namespace
Add Type
Delete
Preview

```
System.Reflection.AssemblyCompanyAttribute
System.Reflection.AssemblyConfigurationAttribute
System.Reflection.AssemblyCopyrightAttribute
System.Reflection.AssemblyProductAttribute
System.Reflection.AssemblyTitleAttribute
System.Reflection.AssemblyTrademarkAttribute
HelloWorld.exe
  HelloWorld
    Friendly
      .ctor : void(string)
      SayGoodbye : void(string)
      SayHello : void()
      myName : string
      Name
        get_Name : string()
        set_Name : void(string)
    Hello
      .ctor : void()
      Converse : void(string,string)
      Main : void(string[])
```

```
.*
  [regex="true" excludetype="true" speclist=".*" applytoderivedtypes="true"]
  S.*
    [regex="true" speclist=".*" signature=".*"]
```

☑ Regular Expression ☐ Exclude Type
Name: S.*
Signature: .*
Description:

In the screen shot, a method rule is defined that selects the names of all public methods (in all types) whose names start with "**S**".

2.5.9.2.4 Selecting By Field

Field rules are qualified by type rules, so they appear in the rules view as children of type nodes. A field rule will select all fields (in all types matched by the parent type rule) that match your criteria. Supported matching criteria include field name and field attributes.

Field Name

You create a field rule by right clicking on the parent type rule's node and selecting **Add Field**, then typing a name in the *Name* field. The name will be interpreted as a regular expression if the **Regular Expression** checkbox is checked; otherwise the name will be interpreted literally.

Field Attribute Specifier

In addition to field name, you can also select based on field attribute specifiers, using the values provided in the *Attribute Specifier* list box. A '**-**' preceding an attribute specifier negates the attribute (*i.e.* it selects all fields that do not have the specified attribute). You can select multiple attributes from the list; the criteria implied by multiple selections are logically **AND**-ed together (that is, the set of selected fields is the intersection of all fields that match each attribute specifier.). For example, you can select fields that are both public and static by selecting **+public** and **+static** from the list.

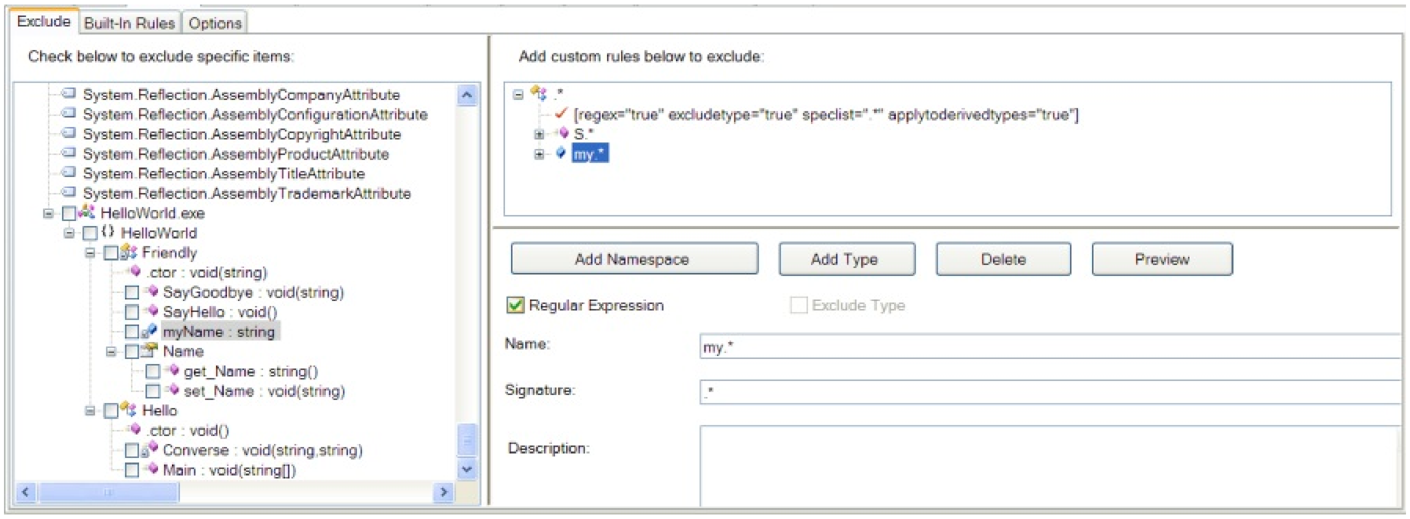
The attribute specifications are logically **AND**-ed with the field name, so if you want to select all fields with a specific set of attributes, you need to provide a regular expression for the field name that selects all fields (*i.e.* ".*").

Field Signature

You can also select fields by signature. A signature specifies the type of the field. The field signature reduces the scope of the field rule, so if you want to create a rule that selects fields regardless of type, you need to provide a regular expression for the signature that selects all signatures (*i.e.* ".*****"). This is the default value.

Field Rule Node

The corresponding field node displayed in the rule editing view has a child element that indicates whether the rule is a regular expression and whether the rule has attribute specifiers, and/or a signature associated with it. You can preview the fields selected by the rule by right clicking on the node and selecting the **Preview** option from the menu.


Exclude
Built-In Rules
Options
Check below to exclude specific items:
Add custom rules below to exclude:
Add Namespace
Add Type
Delete
Preview

```
System.Reflection.AssemblyCompanyAttribute
System.Reflection.AssemblyConfigurationAttribute
System.Reflection.AssemblyCopyrightAttribute
System.Reflection.AssemblyProductAttribute
System.Reflection.AssemblyTitleAttribute
System.Reflection.AssemblyTrademarkAttribute
HelloWorld.exe
  HelloWorld
    Friendly
      .ctor : void(string)
      SayGoodbye : void(string)
      SayHello : void()
      myName : string
      Name
        get_Name : string()
        set_Name : void(string)
    Hello
      .ctor : void()
      Converse : void(string,string)
      Main : void(string[])
```

```
.*
  [regex="true" excludetype="true" speclist="-*" applytoderivedtypes="true"]
  S.*
  my.*
```

Regular Expression [✓]
Exclude Type [ ]
Name: my.*
Signature: .*
Description:

In the screen shot, a field rule is defined that selects the names of all fields (in all types) with names that start with "**my**".

2.5.9.2.5 Selecting By Property

Property rules are qualified by type rules, so they appear in the rules view as children of type nodes. A property rule will select all properties (in all types matched by the parent type rule) that match your criteria. Supported matching criteria include property name and property attributes.

Property Name

You create a property rule by right clicking on the parent type rule's node and selecting **Add Property**, then typing a name in the *Name* field. The name will be interpreted as a regular expression if the **Regular Expression** checkbox is checked; otherwise the name will be interpreted literally.

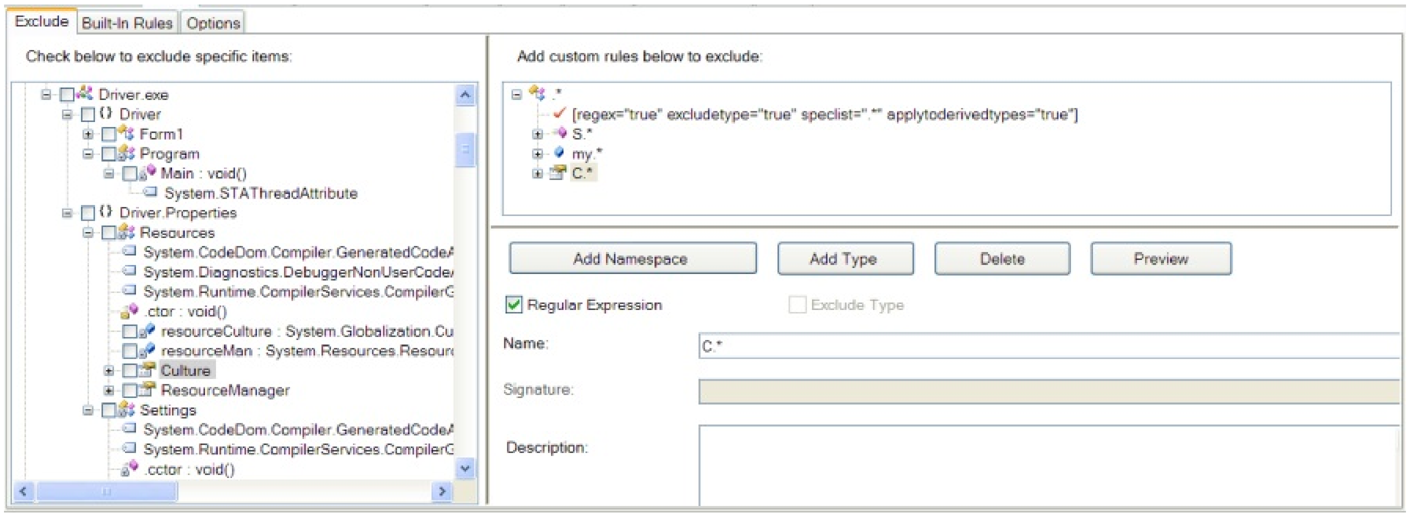
Property Attribute Specifier

In addition to property name, you can also select based on property attribute specifiers, using the values provided in the *Attribute Specifier* list box. A '**-**' preceding an attribute specifier negates the attribute (*i.e.* it selects all properties that do not have the specified attribute). You can select multiple attributes from the list; the criteria implied by multiple selections are logically **`AND`**-ed together (that is, the set of selected properties is the intersection of all methods that match each attribute specifier.). For example, you can select properties that are both public and virtual by selecting **+public** and **+virtual** from the list.

The attribute specifications are logically **`AND`**-ed with the property's name and signature list, so if you want to select all properties with a specific set of attributes, you need to provide a regular expression for the property name that selects all properties (*i.e.* ".*").

Property Rule Node

The corresponding property node displayed in the rule editing view has a child element that indicates whether the rule is a regular expression and whether the rule has attribute specifiers. You can preview the items selected by the rule by right clicking on the node and selecting the **Preview** option from the menu.


Exclude
Built-In Rules
Options
Check below to exclude specific items:
Driver.exe
Driver
Form1
Program
Main : void()
System.STAThreadAttribute
Driver.Properties
Resources
System.CodeDom.Compiler.GeneratedCode
System.Diagnostics.DebuggerNonUserCode
System.Runtime.CompilerServices.Compiler
.ctor : void()
resourceCulture : System.Globalization.Cu
resourceMan : System.Resources.Resour
Culture
ResourceManager
Settings
System.CodeDom.Compiler.GeneratedCode
System.Runtime.CompilerServices.Compiler
.cctor : void()
Add custom rules below to exclude:
Add Namespace
Add Type
Delete
Preview
Exclude Type
Signature:
Description:

```
.*
  [regex="true" excludetype="true" speclist="+*" applytoderivedtypes="true"]
  S.*
  my.*
  C.*
```

[✓] Regular Expression

Name: C.*

In the screen shot, a property rule is defined that selects the names of all public properties (in all types) whose names start with "**C**".

2.5.9.2.6 Selecting By Event

Event rules are qualified by type rules, so they appear in the rules view as children of type nodes. An event rule will select all events (in all types matched by the parent type rule) that match your criteria. Supported matching criteria include event name and event attributes.

Event Name

You create a event rule by right clicking on the parent type rule's node and selecting **Add Event**, then typing a name in the *Name* field. The name will be interpreted as a regular expression if the **Regular Expression** checkbox is checked; otherwise the name will be interpreted literally.

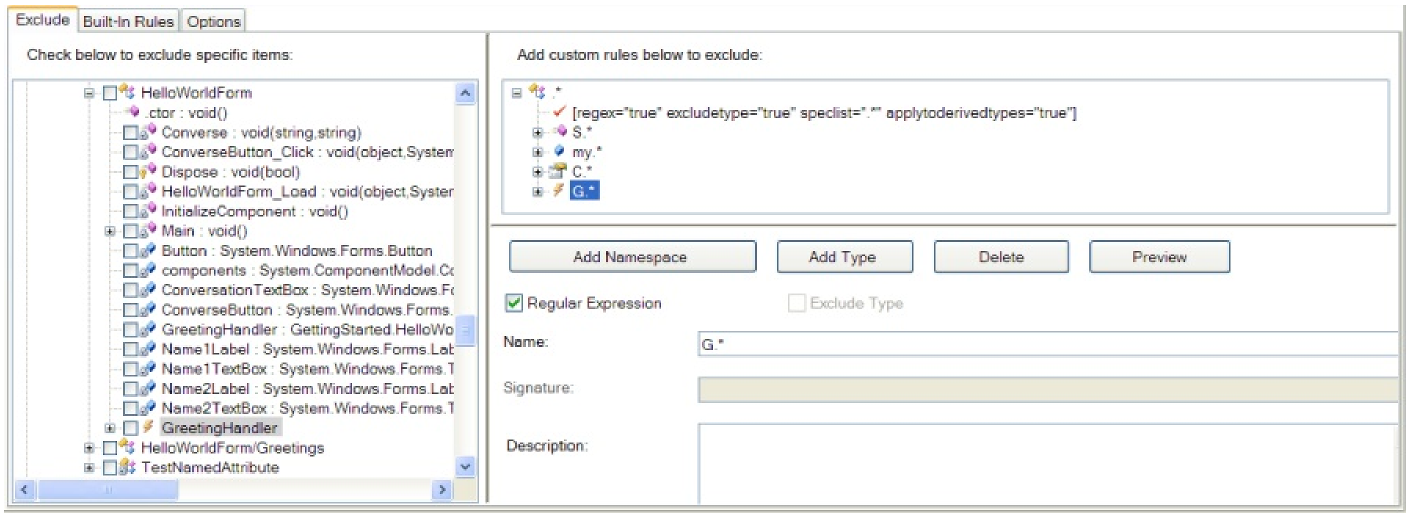
Event Attribute Specifier

In addition to event name, you can also select based on event attribute specifiers, using the values provided in the *Attribute Specifier* list box. A '**-**' preceding an attribute specifier negates the attribute (*i.e.* it selects all events that do not have the specified attribute). You can select multiple attributes from the list; the criteria implied by multiple selections are logically **`AND`**-ed together (that is, the set of selected events is the intersection of all events that match each attribute specifier.). For example, you can select events that are both public and static by selecting **+public** and **+static** from the list.

The attribute specifications are logically **`AND`**-ed with the event name, so if you want to select all events with a specific set of attributes, you need to provide a regular expression for the event name that selects all events (*i.e.* ".*").

Event Rule Node

The corresponding event node displayed in the rule editing view has a child element that indicates whether the rule is a regular expression and whether the rule has attribute specifiers. You can preview the events selected by the rule by right clicking on the node and selecting the **Preview** option from the menu.


Exclude
Built-In Rules
Options
Check below to exclude specific items:
Add custom rules below to exclude:
Add Namespace
Add Type
Delete
Preview

```
HelloWorldForm
  .ctor : void()
  Converse : void(string,string)
  ConverseButton_Click : void(object,System
  Dispose : void(bool)
  HelloWorldForm_Load : void(object,Syster
  InitializeComponent : void()
  Main : void()
  Button : System.Windows.Forms.Button
  components : System.ComponentModel.Co
  ConversationTextBox : System.Windows.Fo
  ConverseButton : System.Windows.Forms.
  GreetingHandler : GettingStarted.HelloWo
  Name1Label : System.Windows.Forms.Lab
  Name1TextBox : System.Windows.Forms.T
  Name2Label : System.Windows.Forms.Lab
  Name2TextBox : System.Windows.Forms.T
  GreetingHandler
HelloWorldForm/Greetings
TestNamedAttribute
```

```
.*
  [regex="true" excludetype="true" speclist=".*" applytoderivedtypes="true"]
  S.*
  my.*
  C.*
  G.*
```

[✓] Regular Expression [ ] Exclude Type

Name: G.*

Signature:

Description:

In the screen shot, an event rule is defined that selects the names of all events (in all types) with names that start with "**G**".

2.5.9.2.7 Selecting By Custom Attribute

Custom attribute rules are qualified by type, method, field, property, or event rules, so they appear in the rules view as children of type, method, field, property, or event nodes. A custom attribute rule will select all items selected by the parent node that are also annotated with a matching custom attribute.

Custom Attribute Name

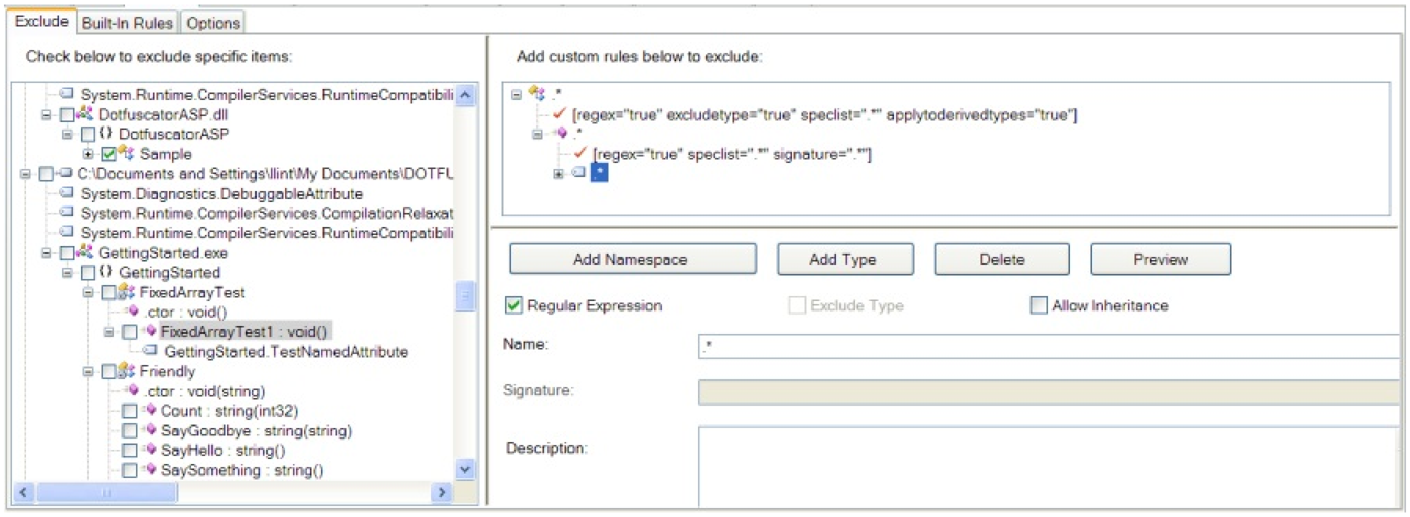
You create a custom attribute rule by right clicking on the parent type, method, field, property, or event rule's node and selecting **Add Custom Attribute**, then typing a name in the *Name* field. The name will be interpreted as a regular expression if the **Regular Expression** checkbox is checked; otherwise the name will be interpreted literally.

Allow Inheritance Checkbox

The **Allow Inheritance** checkbox controls how the custom attribute rule is applied to inheritance hierarchies. If checked, the rule will additionally exclude overriding methods, properties, events, and sub types.

Custom Attribute Rule Node

The corresponding custom attribute node displayed in the rule editing view has a child element that indicates whether the rule is a regular expression. You can preview the types, methods, fields, properties, or events selected by the rule by right clicking on the node and selecting the **Preview** option from the menu.


Exclude
Built-In Rules
Options
Check below to exclude specific items:
System.Runtime.CompilerServices.RuntimeCompatibili
DotfuscatorASP.dll
DotfuscatorASP
Sample
C:\Documents and Settings\llint\My Documents\DOTFL
System.Diagnostics.DebuggableAttribute
System.Runtime.CompilerServices.CompilationRelaxat
System.Runtime.CompilerServices.RuntimeCompatibili
GettingStarted.exe
GettingStarted
FixedArrayTest
.ctor : void()
FixedArrayTest1 : void()
GettingStarted.TestNamedAttribute
Friendly
.ctor : void(string)
Count : string(int32)
SayGoodbye : string(string)
SayHello : string()
SaySomething : string()
Add custom rules below to exclude:
Add Namespace
Add Type
Delete
Preview

```
.*
  [regex="true" excludetype="true" speclist=".*" applytoderivedtypes="true"]
  .*
    [regex="true" speclist=".*" signature=".*"]
    .*
```

[✓] Regular Expression [ ] Exclude Type [ ] Allow Inheritance

Name: .*

Signature:

Description:

In the screen shot, a custom attribute rule is defined that selects all methods that are annotated with a custom attribute named **GettingStarted.TestNamedAttribute**.

2.5.9.2.8 Selecting By Supertype

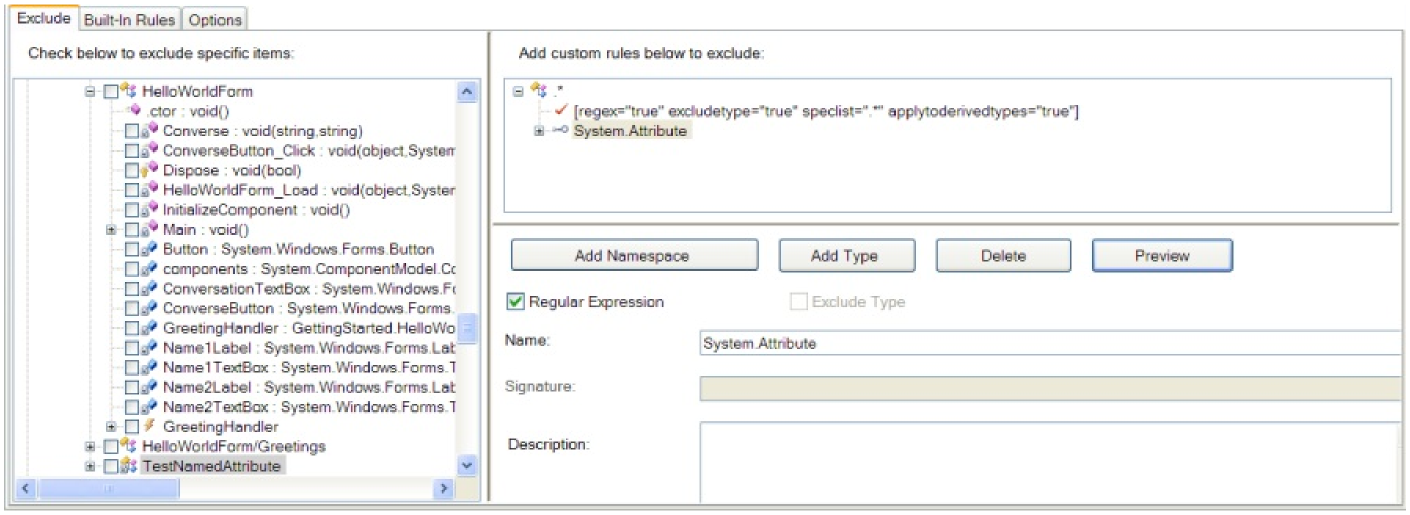
Supertype rules are qualified by type rules, so they appear in the rules view as children of type nodes. A supertype rule will narrow the scope of a type rule so that only types matched by the parent type rule that also derive from the specified supertype are selected.

Supertype Name

You create a supertype rule by right clicking on the parent type rule's node and selecting **Add Supertype**, then typing a name in the *Name* field. The name will be interpreted as a regular expression if the **Regular Expression** checkbox is checked; otherwise the name will be interpreted literally.

Supertype Rule Node

The corresponding supertype node displayed in the rule editing view has a child element that indicates whether the rule is a regular expression. You can preview the types selected by the rule by right clicking on the node and selecting the **Preview** option from the menu.



Check below to exclude specific items:

```
HelloWorldForm
    .ctor : void()
    Converse : void(string,string)
    ConverseButton_Click : void(object,Systen
    Dispose : void(bool)
    HelloWorldForm_Load : void(object,Syster
    InitializeComponent : void()
    Main : void()
    Button : System.Windows.Forms.Button
    components : System.ComponentModel.Co
    ConversationTextBox : System.Windows.Fo
    ConverseButton : System.Windows.Forms.
    GreetingHandler : GettingStarted.HelloWo
    Name1Label : System.Windows.Forms.Lab
    Name1TextBox : System.Windows.Forms.T
    Name2Label : System.Windows.Forms.Lab
    Name2TextBox : System.Windows.Forms.T
    GreetingHandler
HelloWorldForm/Greetings
TestNamedAttribute
```

Add custom rules below to exclude:

```
.*
    [regex="true" excludetype="true" speclist=".*" applytoderivedtypes="true"]
    System.Attribute
```

[✓] Regular Expression
[ ] Exclude Type

Name: System.Attribute

Signature:

Description:

In the screen shot, a supertype rule is defined that selects all types that are supertypes of **System.Attribute**.

2.5.9.3 Editing and Deleting Rules

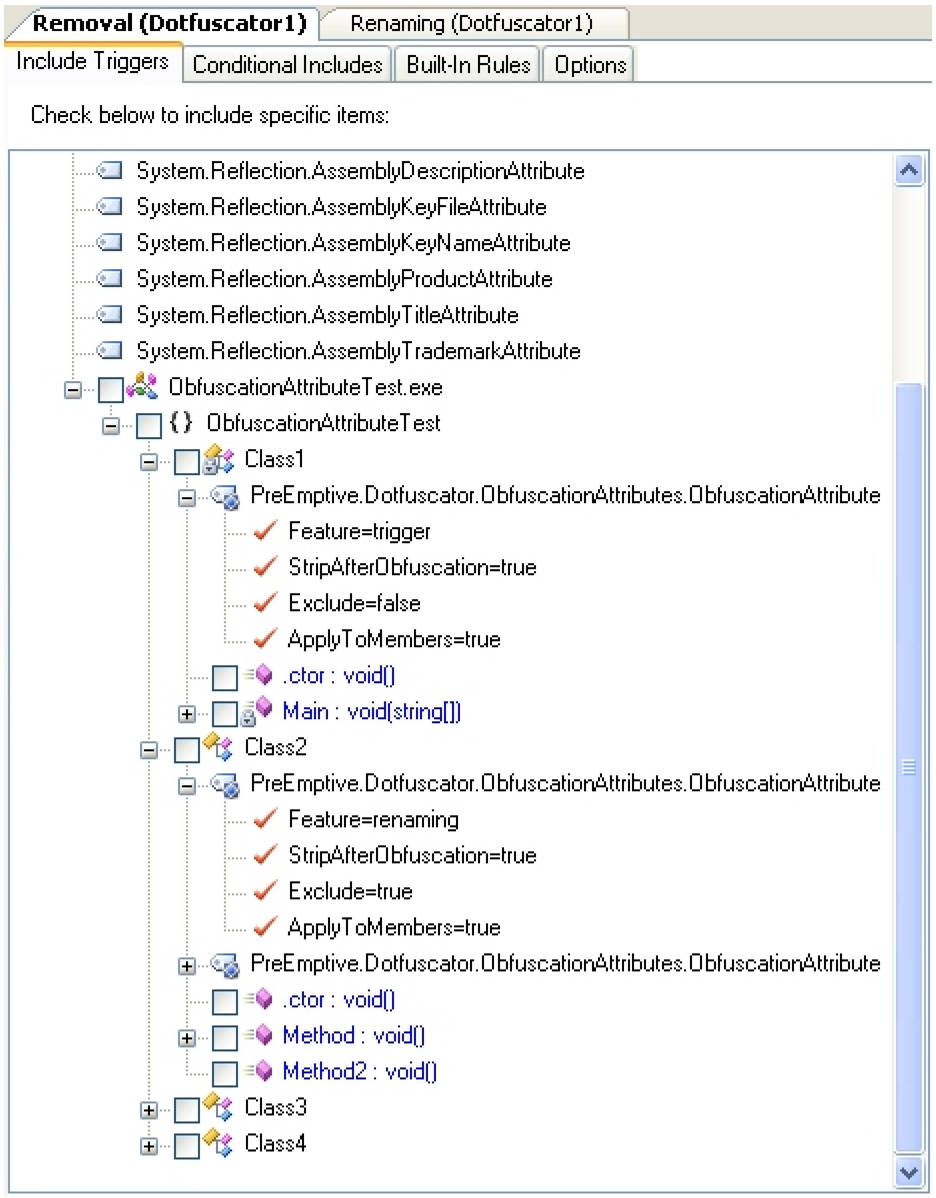
To edit an existing rule, simply click on the rule in the rule editing view. You can then use controls below the view to edit the values associated with the node (e.g. name, attribute specifier list, etc.).

To delete a rule, click on the rule in the rule editing view, and press the **Delete** button.

2.5.9.4 Using Declarative Obfuscation with Rules

The rules editor provides support for Declarative Obfuscation by displaying the arguments of all obfuscation attributes (*i.e.* `System.Reflection.ObfuscateAssemblyAttribute` and `System.Reflection.ObfuscationAttribute`) in the application tree view. Items in the application tree view (types, methods, fields) that are selected by an obfuscation attribute appear in blue.

In the screenshot below, the methods of Class1 and Class2 are marked as removal triggers using `ObfuscationAttributes`. Each attribute's properties and values are expanded in the view.



Check below to include specific items:

```
System.Reflection.AssemblyDescriptionAttribute
System.Reflection.AssemblyKeyFileAttribute
System.Reflection.AssemblyKeyNameAttribute
System.Reflection.AssemblyProductAttribute
System.Reflection.AssemblyTitleAttribute
System.Reflection.AssemblyTrademarkAttribute
ObfuscationAttributeTest.exe
  ObfuscationAttributeTest
    Class1
      PreEmptive.Dotfuscator.ObfuscationAttributes.ObfuscationAttribute
        Feature=trigger
        StripAfterObfuscation=true
        Exclude=false
        ApplyToMembers=true
      .ctor : void()
      Main : void(string[])
    Class2
      PreEmptive.Dotfuscator.ObfuscationAttributes.ObfuscationAttribute
        Feature=renaming
        StripAfterObfuscation=true
        Exclude=true
        ApplyToMembers=true
      PreEmptive.Dotfuscator.ObfuscationAttributes.ObfuscationAttribute
      .ctor : void()
      Method : void()
      Method2 : void()
    Class3
    Class4
```

2.5.9.5 Previewing Rules

To preview the effects of a single rule in the rule editing view, right click on the rule's node and select **Preview** from the menu. Items selected by the rule will appear shaded in the application tree view.

To preview the combined effects of all the rules defined in the rule editing view, click on the **Preview** button. All items selected by applying all the rules will appear shaded in the application tree view.

2.5.10 Instrumentation (Tamper, Shelf Life, Exception, Analytics)

The *Instrumentation* Tab allows you to add, edit, and review the custom attributes that configure Dotfuscator's code-injection features (called *Instrumentation*). Those features include:

- Shelf Life
- Tamper Notification
- Exception Tracking
- PreEmptive Analytics

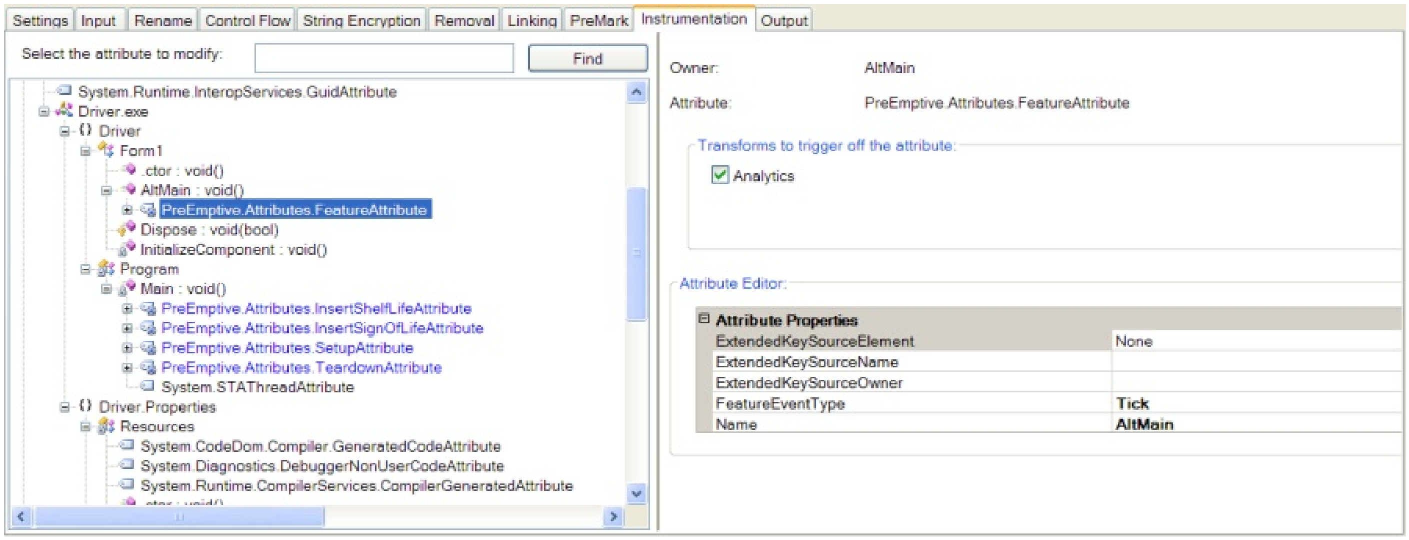
For new projects, the default setting for the Instrumentation transform is **Enabled**.

Supported custom attributes already in your source code can be edited through the user interface. Changes are persisted to the Dotfuscator configuration file and take precedence over the attributes in source code.

Extended custom attributes can also be added and edited through the user interface. Dotfuscator treats extended attributes the same as it treats custom attributes embedded in the source code.

You can also map supported attributes (even attributes embedded in code) to a particular supported code transform. This supports attribute overloading, wherein the same set of attributes can drive multiple transforms (e.g. Application Analytics).

To add an attribute, navigate to the method or assembly you wish to place the attribute on and select it. If the attribute has arguments, you will be able to set them in the instrumentation editor.



```
System.Runtime.InteropServices.GuidAttribute
Driver.exe
  {} Driver
    Form1
      .ctor : void()
      AltMain : void()
        PreEmptive.Attributes.FeatureAttribute
      Dispose : void(bool)
      InitializeComponent : void()
    Program
      Main : void()
        PreEmptive.Attributes.InsertShelfLifeAttribute
        PreEmptive.Attributes.InsertSignOfLifeAttribute
        PreEmptive.Attributes.SetupAttribute
        PreEmptive.Attributes.TeardownAttribute
        System.STAThreadAttribute
  {} Driver.Properties
    Resources
      System.CodeDom.Compiler.GeneratedCodeAttribute
      System.Diagnostics.DebuggerNonUserCodeAttribute
      System.Runtime.CompilerServices.CompilerGeneratedAttribute
```

Owner: AltMain

Attribute: PreEmptive.Attributes.FeatureAttribute

Transforms to trigger off the attribute:

[✓] Analytics

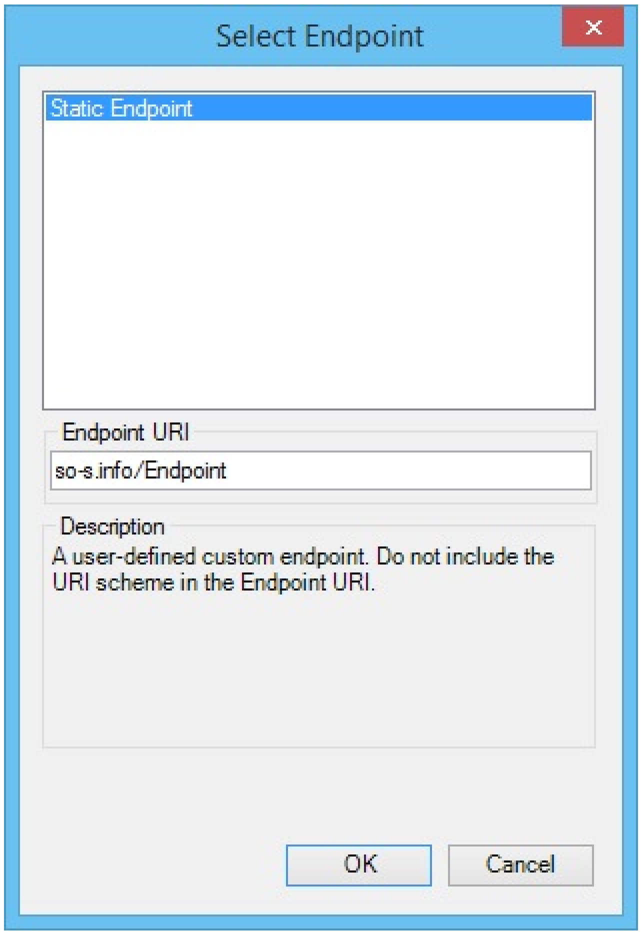
Attribute Editor:

Attribute Properties	
ExtendedKeySourceElement	None
ExtendedKeySourceName	
ExtendedKeySourceOwner	
FeatureEventType	**Tick**
Name	**AltMain**

Checking the **Analytics** checkbox in the *Transforms to trigger off the attribute:* section instruments the attributed method as a feature to be tracked with PreEmptive Analytics.

Custom Endpoint

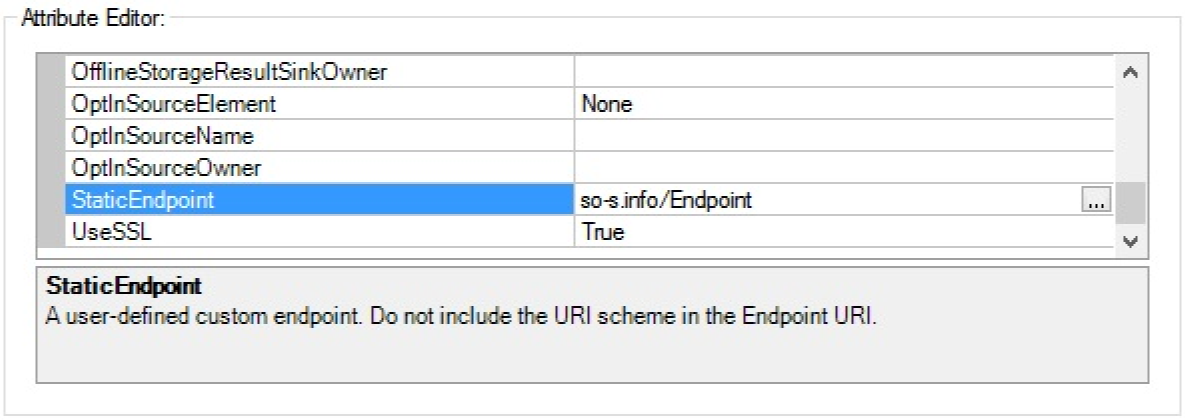
In the *Attribute Editor* for the SetupAttribute, the *StaticEndpoint* field is where you explicitly specify the endpoint for PreEmptive Analytics messages. If no custom endpoint is selected, PreEmptive's Commercial Runtime Intelligence Services endpoint will be used. Click the **"..."** (ellipses) in the *StaticEndpoint* field to invoke the Select Endpoint window:



so-s.info/Endpoint

A user-defined custom endpoint. Do not include the URI scheme in the Endpoint URI.

Here you are able to specify what endpoint you wish to send to. Once you've made your selection and clicked OK, the location displays in the StaticEndpoint field:

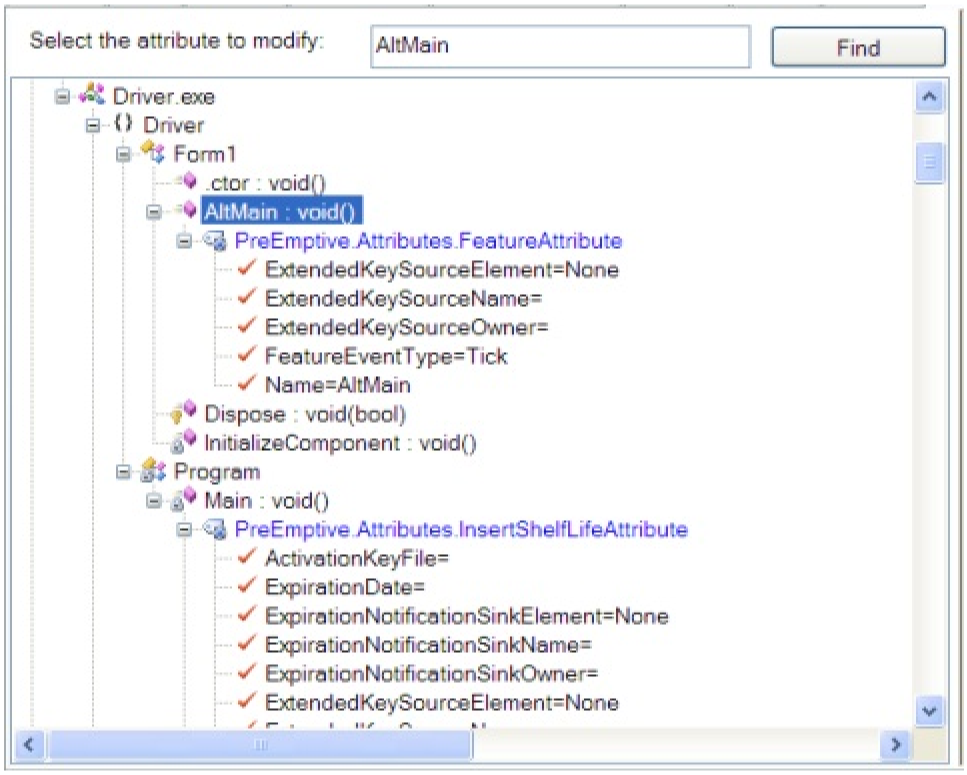

Attribute Editor:

OfflineStorageResultSinkOwner	
OptInSourceElement	None
OptInSourceName	
OptInSourceOwner	
StaticEndpoint	so-s.info/Endpoint
UseSSL	True

StaticEndpoint
A user-defined custom endpoint. Do not include the URI scheme in the Endpoint URI.

Attribute Search Field

The *Instrumentation* tab is enabled with a **Find** feature that enables you to easily locate any item in the instrumentation tree view. You may type the full name or the first few letters of any item you wish to find in the *Select the attribute to modify:* field then click **Find**. You may also search for an item by typing **.*** (the wildcard) after the first few letters of the name, or you may enter a Regular Expression. Dotfuscator will select the first item where your search term appears. Clicking on the **Find** button will select the next match for your search term. Please note that since the search matches the beginning of an item's name, when searching for instrumentation attributes you will need to search using the full name of the attribute. For example, to find a setup attribute, enter *PreEmptive.Attributes.SetupAttribute.*


Select the attribute to modify: AltMain

```
Driver.exe
  Driver
    Form1
      .ctor : void()
      AltMain : void()
        PreEmptive.Attributes.FeatureAttribute
          ExtendedKeySourceElement=None
          ExtendedKeySourceName=
          ExtendedKeySourceOwner=
          FeatureEventType=Tick
          Name=AltMain
      Dispose : void(bool)
      InitializeComponent : void()
    Program
      Main : void()
        PreEmptive.Attributes.InsertShelfLifeAttribute
          ActivationKeyFile=
          ExpirationDate=
          ExpirationNotificationSinkElement=None
          ExpirationNotificationSinkName=
          ExpirationNotificationSinkOwner=
          ExtendedKeySourceElement=None
```

2.5.10.1 Shelf Life Token Overview

Shelf Life is an application inventory management function that allows you to embed expiration and notification logic into an application. Dotfuscator injects code that reacts to application expiration by exiting the application and/or sending a PreEmptive Analytics message. This feature is particularly helpful with beta applications. Users can schedule an application's expiration/de-activation for a specific date and optionally issue warnings to users that the application will expire/de-activate in a specific number of days.

A Shelf Life Token is an encrypted set of data containing application and expiration information that is injected into the application binary at obfuscation time, or it can be stored as a string outside of the application binary so that it can be updated to extend the expiration date.

Using a Private Key File and password is an option that provides additional defense in depth of a Shelf Life token by signing the token data with the user provided private key. Only the necessary public key information necessary to provide signature validation is stored in the Shelf Life Token.

There are two methods of providing the Shelf Life Token to the Shelf Life runtime code that is injected into the application. The default method is for Dotfuscator to directly embed the encrypted data into the binary during the obfuscation process. An alternative method is for the application developer to provide for their own persistence of the Shelf Life Token data and to provide a string representation of that Shelf Life Token data to the injected runtime code via custom methods, properties, etc. as set in the ShelfLifeTokenSource properties. The advantage of persisting the Shelf Life Token data outside of the binary allows for easier extensions of expiration dates without distributing a newly instrumented binary but, it does so at the cost of making the Shelf Life Token Data more visible to the end user.

2.5.10.1.1 Shelf Life Activation Key Overview

A Shelf Life Activation Key (SLAK**)** is a data file that is required to inject Shelf Life functionality into Dotfuscator and into the appropriate locations within an application that you want to instrument.

A Shelf Life Activation Key is issued by PreEmptive and provided to Dotfuscator by the user during shelf life configuration. To obtain a Shelf Life Activation Key, contact PreEmptive Solutions. PreEmptive will issue you a data file containing the Shelf Life Activation Key that is to be stored on your Build Machine.

Once the Shelf Life Activation Key is obtained, you can add the Shelf Life Attribute to a method or group of methods. In the *Instrumentation* tab in the *Attribute Editor*: section, in the *ActivationKeyFile* field you must select the path to the Shelf Life Activation Key file, thereby activating Shelf Life within your application.

2.5.10.1.2 Expiration and Warning Actions

Users can specify what action results when expiration or warning occur via configuration or custom attribute. If the user selects the default action, the application exits upon expiration and takes no action during the warning period. The **`InsertShelfLife`** attribute can be used to inject code to perform a call back to a user-specified method when a Shelf Life warning or expiration occurs.

Expiration Notification

To accomplish this, the **`InsertShelfLife`** attribute is used to specify an **`ExpirationNotificationSink`**, which Dotfuscator uses to generate code that communicates the results of the shelf life expiration check back to the application. The **`ExpirationNotificationSink`** may be a writeable boolean valued property or field, or it may be a method or delegate with the signature **`void(bool)`** or **`void(string,string)`**. After a shelf life expiration check, the generated code sets the boolean value to true if the application is expired; false if not. If using a method or delegate sink with a **`void(string,string)`** signature, the generated code will call this sink and will pass in the warning date and the expiration date as the string arguments, in that order. The application is free to react in any way in response to a shelf life expiration notification. The **`InsertShelfLife`** attribute defines three properties for specifying an **`ExpirationNotificationSink`**:

- **ExpirationNotificationSinkElement.**
- **ExpirationNotificationSinkOwner.**
- **ExpirationNotificationSinkName**.

These properties are described in detail in the InsertShelfLifeAttribute section of the custom attribute reference.

The expiration notification sink settings are optional. If they are omitted, the application is not notified when the expiration check is executed.

If the **`ExpirationNotificationSinkElement`** is set to **DefaultAction**, Dotfuscator injects code that exits the application if shelf life expiration is detected.

Warning Notification

To accomplish this, the **`InsertShelfLife`** attribute is used to specify a **`WarningNotificationSink`**, which Dotfuscator uses to generate code that communicates the results of the shelf life warning check back to the application. The **`WarningNotificationSink`** may be a writeable boolean valued property or field, or it may be a method or delegate with the signature **`void(bool)`** or **`void(string,string)`**. After a shelf life warning check, the generated code sets the boolean value to true if the application is in its warning period; false if it is not. If using a method or delegate sink with a **`void(string,string)`** signature, the generated code will call this sink and will pass in the warning date and the expiration date as the string arguments, in that order. The application is free to react in any way in response to a shelf life warning notification. The **`InsertShelfLife`** attribute defines three properties for specifying a **`WarningNotificationSink`**:

- **WarningNotificationSinkElement.**
- **WarningNotificationSinkOwner.**
- **WarningNotificationSinkName**.

These properties are described in detail in the InsertShelfLifeAttribute section of the custom attribute reference.

The warning notification sink settings are optional. If they are omitted, the application is not notified when the warning check is executed.

If the **`WarningNotificationSinkElement`** is set to **DefaultAction**, Dotfuscator does not inject any code to handle the warning notification.

Sample **`InsertShelfLife`** attribute usage with **`ExpirationNotificationSink`** and **`WarningNotificationSink`** as instance fields defined in the same class as the attributed method:

InsertShelfLife Attribute Usage with WarningNotificationSink and ExpirationNotificationSink

```
class ShelfLifeSample {
   bool instanceShelfLifeExpirationFlag;

   [PreEmptive.Attributes.InsertShelfLife(
      ActivationKeyFile = "C:\\shelflife.slkey",
      ExpirationDate = "2009-11-05",
      ExpirationNotificationSinkElement = SinkElements.Field,
      ExpirationNotificationSinkName = "instanceShelfLifeExpirationFlag",
      WarningDate = "2009-07-04",
      WarningNotificationSinkElement = SinkElements.Method,
      WarningNotificationSinkName = "CheckShelfLifeWarning"
   )]
   private void Verify() {
     // Dotfuscator will add Shelf Life expiration and warning detection and
notification code here
   }
   private CheckShelfLifeExpirationState() {
      if ( instanceShelfLifeExpirationFlag ) {
         // app has expired
      }
      else {
         // app has not expired
      }
   }
   private CheckShelfLifeWarning(string warnDate, string expDate) {
      // use date strings directly
      Console.WriteLine("MyApp expires on " + expDate);

      // or parse to datetime to do calculations
      int daysLeft = DateTime.Parse(expDate).Subtract(DateTime.Now);
      Console.WriteLine("MyApp will expire in " + daysLeft + " days");
   }
}
```

2.5.10.1.3 Expiration and Warning Reporting

The injected Shelf Life code can send messages to a PreEmptive Analytics Endpoint when the warning period is entered or expiration has occurred. To enable this feature, turn on **Send Shelf Life Messages** in the *Global Options* section of the *Settings* tab.

To detect Shelf Life expiration, place **`InsertShelfLife`** attributes on one or more methods in the application that are always executed. When Dotfuscator encounters an **`InsertShelfLife`** attribute during its processing, it adds code that performs expiration detection at runtime. If the current date is on or after than the embedded expiration date then a Shelf Life expiration message is sent to a PreEmptive Analytics Endpoint.

Dotfuscator also allows you to optionally specify a warning period that will occur prior to the application's expiration. If the current date is on or after than the embedded warning date then a Shelf Life warning message is sent to a PreEmptive Analytics Endpoint.

`InsertShelfLife` attributes are not required at runtime; therefore, Dotfuscator strips them from the output application.

An application can contain any number of **`InsertShelfLife`** attributes. In the event that an application has expired or is about to expire, multiple shelf life messages from the same application session will be sent with the same Shelf Life ID.

⚠ Do not put this attribute on the same method containing the Setup Attribute. Methods with this attribute must be executed after the method containing the Setup Attribute.

InsertShelfLife Attribute

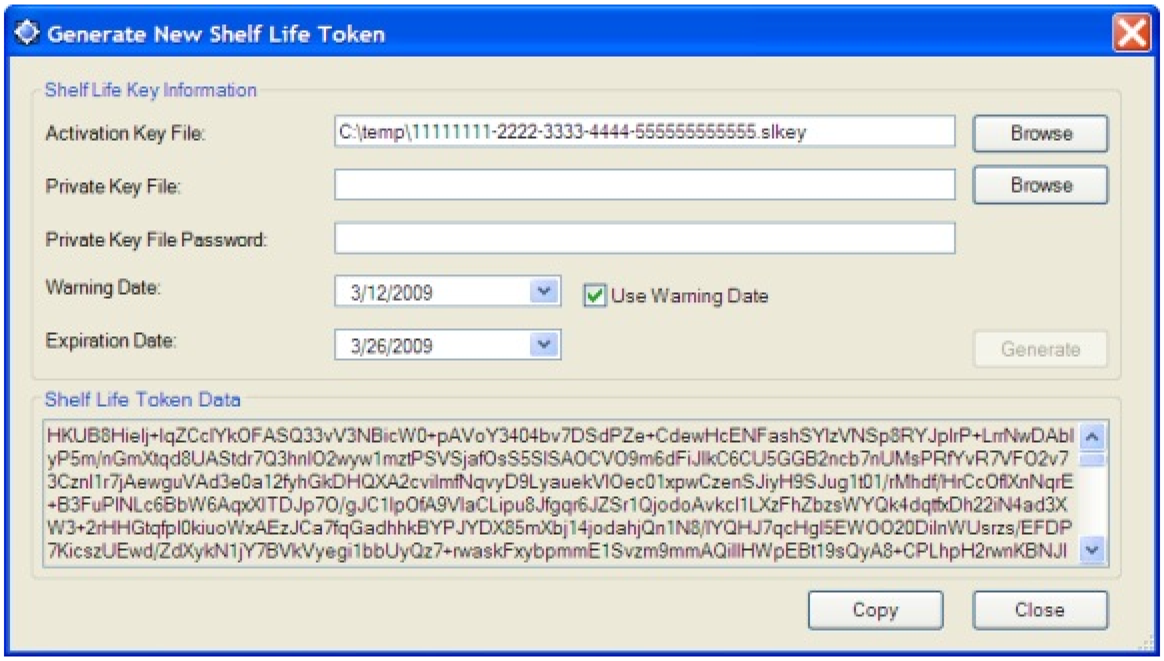
```
[PreEmptive.Attributes.InsertShelfLife(
   ActivationKeyFile = "C:\\shelflife.slkey",
   ExpirationDate = "2009-11-05"
)]
public void DoStuff() { ... }
```

Note: A Shelf Life Activation Key (SLAK) must be purchased separately in order to use this functionality.

2.5.10.1.4 Generate New Shelf Life Token

When using Shelf Life with a Shelf Life Token Source Dotfuscator allows you to generate new Shelf Life Tokens easily by clicking **View > Generate New Shelf Life Token...** in the menu bar of Visual Studio, or by clicking **Tools > Generate Shelf Life Token...** in Dotfuscator.

Within the dialog box that displays, you can browse to and select the appropriate Shelf Life Activation Key file and optionally, a PKCS #12 Private Key file to provide additional validation of the Shelf Life token. When using a private key file, enter the correct **password** in the *Private Key File Password* field. Set the *Expiration Date* and optionally set the *Warning Date*. Next to the *Warning Date* field is the *Use Warning Date* checkbox. Clear this check box if you did not enable Warning Date behavior using the InsertShelfLifeAttribute during instrumentation, or if you enabled it during instrumentation but wish to provide an updated shelf life token that disables it.



Activation Key File: C:\temp\11111111-2222-3333-4444-555555555555.slkey

Private Key File:

Private Key File Password:

Warning Date: 3/12/2009 [✓ Use Warning Date]

Expiration Date: 3/26/2009

HKUB8Hielj+lqZCcIYkOFASQ33vV3NBicW0+pAVoY3404bv7DSdPZe+CdewHcENFashSYlzVNSp8RYJplrP+LrrNwDAbl
yP5m/nGmXtqd8UAStdr7Q3hnlO2wyw1mztPSVSjafOsS5SISAOCVO9m6dFiJlkC6CU5GGB2ncb7nUMsPRfYvR7VFO2v7
3Cznl1r7jAewguVAd3e0a12fyhGkDHQXA2cvilmfNqvyD9LyauekVlOec01xpwCzenSJiyH9SJug1t01/rMhdf/HrCcOfXnNqrE
+B3FuPlNLc6BbW6AqxXITDJp7O/gJC1lpOfA9VlaCLipu8Jfgqr6JZSr1QjodoAvkcl1LXzFhZbzsWYQk4dqtfxDh22iN4ad3X
W3+2rHHGtqfpl0kiuoWxAEzJCa7fqGadhhkBYPJYDX85mXbj14jodahjQn1N8/lYQHJ7qcHgl5EWOO20DilnWUsrzs/EFDP
7KicszUEwd/ZdXykN1jY7BVkVyegi1bbUyQz7+rwaskFxybpmmE1Svzm9mmAQillHWpEBt19sQyA8+CPLhpH2rwnKBNJI

When the Shelf Life Key information is determined to be correct, the **Generate** button is activated and ready to be clicked. Clicking this button generates a new Shelf Life Token that can be used by a Shelf Life instrumented application via the ShelfLifeTokenSource properties of the InsertShelfLifeAttribute. The Shelf Life Token Data can be copied to the clipboard by clicking the **Copy** button.

2.5.10.2 Tamper Notification

Dotfuscator can instrument applications to detect if they have been tampered with and if so, optionally send a message to a PreEmptive Analytics Endpoint.

When run on a properly attributed .NET application, Dotfuscator processes the Tamper Notification attributes and instruments the application accordingly. The resulting output application will be ready to send Tamper Notifications to a PreEmptive Analytics Endpoint. The only differences between Tamper Notification and Application Analytics at this level are in the attributes.

Tamper Notification Message Type

Tamper Notification defines one message type:

- Tamper Detected

The PreEmptive Analytics Service defines a **Tamper Detected** message. An application sends this message when it determines that it has been modified since being run through Dotfuscator. To have your application send these messages, you must:

- Attribute your application with PreEmptive Analytics attributes, including Setup and Teardown.
- Add InsertTamperCheck attributes to methods where you would like application integrity checks to be performed.

- Run your application through Dotfuscator with the “Send Tamper Messages” option turned on. See Configuring and Running Dotfuscator with Application Analytics.

See Example PreEmptive Analytics Enabled Application to see the contents of messages containing PreEmptive Analytics data.

2.5.10.2.1 Tamper Reporting

To detect tampering, place **`InsertTamperCheck`** attributes on one or more methods in the application that are always executed. When Dotfuscator encounters an **`InsertTamperCheck`** attribute during its processing, it adds code that performs an assembly level integrity check at runtime. If the integrity check fails, it sends a tamper detected message to a PreEmptive Analytics Endpoint. It also calls code that exits the application or any other application defined code (see Tamper Actions). **`InsertTamperCheck`** attributes are not required at runtime; therefore, Dotfuscator strips them from the output application.

An application can contain any number of **`InsertTamperCheck`** attributes. In the event that an application has been tampered with, multiple tamper detected messages from the same application session will be sent with the same group Id.

> ⚠ Do not put this attribute on the same method containing the Setup Attribute. Methods with this attribute must be executed after the method containing the Setup Attribute.

InsertTamperCheck Attribute

```
[PreEmptive.Attributes.InsertTamperCheck()]
public void DoStuff() { ... }
```

2.5.10.2.2 Tamper Actions

In addition to sending tamper notifications to the managed service, the **`InsertTamperCheck`** attribute can be used to inject code that gets executed during the tamper verification.

Application Notification

To accomplish this, the **`InsertTamperCheck`** attribute is used to specify an **`ApplicationNotificationSink`**, which Dotfuscator uses to generate code that communicates the results of the tamper check back to the application. The **`ApplicationNotificationSink`** may be a writeable boolean valued property or field, or it may be a method or delegate with the signature **`void(bool)`**. After a tamper check, the generated code sets the boolean value to true if the application has been tampered with; false if not. The application is free to react in any way in response to a tamper notification.

The **`InsertTamperCheck`** attribute defines three properties for specifying an ApplicationNotificationSink:

- **ApplicationNotificationSinkElement.**

- **ApplicationNotificationSinkOwner.**
- **ApplicationNotificationSinkName**.

These properties are described in detail in the InsertTamperCheckAttribute section of the custom attribute reference.

The application notification sink settings are optional. If they are omitted, the application is not notified when the tamper check is executed.

If the **`ApplicationNotificationSinkElement`** is set to **DefaultAction**, Dotfuscator injects code that exits the application if tampering is detected.

Sample **`InsertTamperCheck`** attribute usage with **`ApplicationNotificationSink`** as an instance field defined in the same class as the attributed method:

InsertTamperCheck Attribute Usage with ApplicationNotificationSink

```
class TamperSample {
   bool instanceTamperFlag;

   [PreEmptive.Attributes.InsertTamperCheck(
      ApplicationNotificationSinkElement = SinkElements.Field,
      ApplicationNotificationSinkName = "instanceTamperFlag"
   )]
   private void Verify() {
     // Dotfuscator will add Tamper detection and notification code here
   }
   private CheckTamperState() {
      if ( instanceTamperFlag ) {
         // app has been tampered with
      }
      else {
         // app has not been tampered with
      }
   }
}
```

2.5.10.2.3 Simulating Tampering

Dotfuscator ships with a simple command line utility that 'tampers' with an assembly. It is called **TamperTester.exe** and is installed in the same folder as Dotfuscator itself.

TamperTester.exe

```
Usage: tampertester <file_name> [destination folder]
```

By running your Dotfuscator assemblies through this utility, you can test that the tamper notification messages are being generated and sent as expected. You can also test any application code you have written to execute in response to tamper detection.

2.5.10.3 Exception Tracking

This section documents the development process when using Exception Tracking. It describes the Dotfuscator user interface and configuration options that relate to Exception Tracking.

2.5.10.3.1 Exception Reporting

To detect exceptions that occur within a method, place `ExceptionTrack` attributes on the method. To detect exceptions that occur anywhere in an assembly, place `ExceptionTrack` attributes on the assembly.

When Dotfuscator encounters an `ExceptionTrack` attribute during its processing, it adds code that detects exceptions of the configured type:

- **Caught:** Dotfuscator will inject code that tracks exceptions right after they enter a 'catch' block.
- **Thrown:** Dotfuscator will inject code that tracks exceptions right before being thrown by a 'throw' statement.
- **Unhandled (default):** Dotfuscator will inject code that tracks exceptions either at the method level by wrapping the method in a try/catch block and re-throwing the exception, or at the assembly level by registering an UnhandledException event handler on the current AppDomain (for .NET Framework applications) or current Application (for Silverlight applications).

Once an exception is detected, it can be reported to the configured PreEmptive Analytics endpoint by setting the `SendReport` property of the `ExceptionTrack` attribute to true (the default).

Dotfuscator can also be configured to obtain information from the user such as a description of the actions leading to the exception and a contact address that the developer can use to solicit additional information or provide notification of an issue that has been fixed. This information will be attached to the exception report message. To obtain this type of user-provided information, specify a `ReportInfoSource`. For more information on configuring a `ReportInfoSource`, see Collecting User-specified Exception Report Information.

The sending of exception reports will honor the opt-in setting of the user if an `OptInSource` has been configured.

Dotfuscator can be configured to obtain explicit consent from the user to send the exception report message. In this case, the user's explicit consent will override the PreEmptive Analytics opt-in setting if one has been configured. To obtain explicit consent to send the exception report message, specify a `ReportInfoSource`. For more information on configuring a `ReportInfoSource`, see Collecting User-specified Exception Report Information.

The report info source settings are optional. If they are omitted, no user-provided information will be collected, and the sending of the exception report messages will be controlled by the PreEmptive Analytics opt-in setting.

2.5.10.3.2 Exception Actions

In addition to sending exception report messages to the managed service, the **`ExceptionTrack`** attribute can be used to inject code to perform a call back to a user-specified method when an exception is detected.

Exception Notification

To accomplish this, the **`ExceptionTrack`** attribute is used to specify an **`ExceptionNotificationSink`**, which Dotfuscator uses to generate code that forwards the detected Exception object back to the application. The **`ExceptionNotificationSink`** may be a writable field or settable property of type System.Exception, or it may be a method or delegate with the signature **`void(System.Exception)`**. After an exception is detected, the generated code sets the field or property value to the Exception object that was captured. If using a method or delegate sink, the generated code will call this sink with the captured Exception object as the only parameter. The application is free to react in any way in response to a detected exception. The **`ExceptionTrack`** attribute defines three properties for specifying an **`ExceptionNotificationSink`**:

- **ExceptionNotificationSinkElement**.
- **ExceptionNotificationSinkOwner**.
- **ExceptionNotificationSinkName**.

These properties are described in detail in the ExceptionTrack section of the custom attribute reference.

The exception notification sink settings are optional. If they are omitted, no custom action will occur when an exception of the configured type is detected.

Sample **`ExceptionTrack`** attribute usage with **`ExceptionNotificationSink`** as method defined in the same class as the attributed method:

ExceptionTrack usage with ExceptionNotificationSink as method

```
[ExceptionTrack(
   ExceptionNotificationSinkElement = SinkElements.Method,
   ExceptionNotificationSinkName = "Response"
)]
private void Foo() {
  ...
}

// Respond to a detected exception
public void Response(System.Exception e) {
  ...
}
```

2.5.10.3.3 Collecting User-specified Exception Report Information

When using Exception Tracking, Dotfuscator can be configured to obtain explicit user consent and to collect comment and contact information from the user. These are provided in the form of key-value pairs that are read at runtime during the construction of an exception report message.

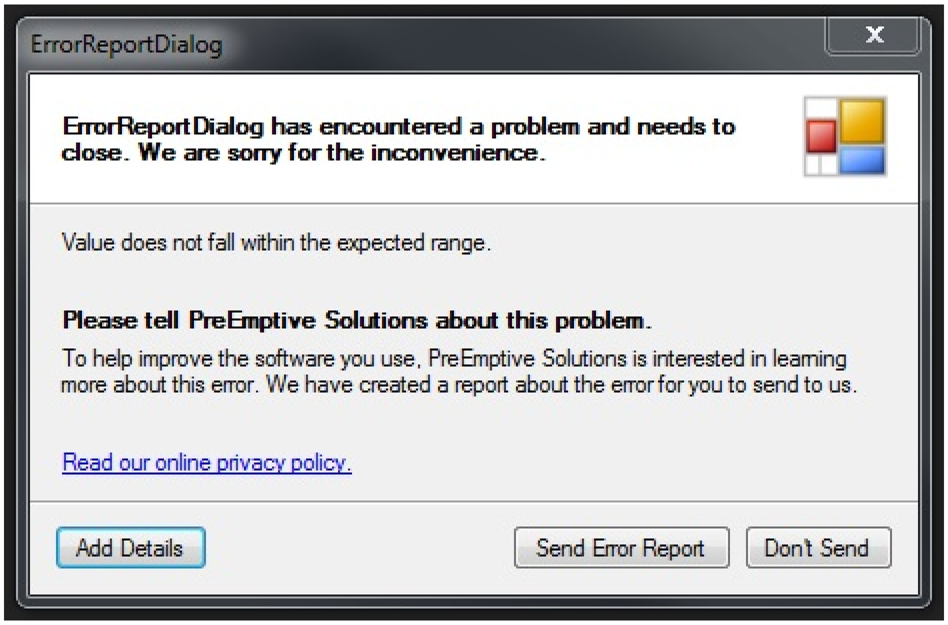
To provide this user-specified report information, specify a **`ReportInfoSource`** on the attribute corresponding to the message you wish to send.

Dotfuscator uses the **`ReportInfoSource`** to generate code that gathers the key-value pairs at runtime. The **`ReportInfoSource`** is an IDictionary or **`IDictionary<string,string>`** valued property, method, field, or when using method-level exception tracking, method argument; it is the developer's responsibility to ensure that a correct values are available in the **`ReportInfoSource`** at the time the an exception is detected.

Using the Built-in Exception Report Dialog as the ReportInfoSource

Dotfuscator can inject a pre-made Exception Report Dialog to ease configuration for most scenarios and provide a consistent user experience for exception reporting. To use the built-in dialog, your assembly must target version 1.1 (or higher) of the .NET Framework, or Silverlight version 2 (or higher). To instruct Dotfuscator to use the built-in dialog as the **`ReportInfoSource`**, set the **`ReportInfoSourceElement`** value to “DefaultAction”.

When using the built-in dialog on the .NET Framework, the dialog will be constructed and displayed using the Windows Forms API. This may have unintended consequences for console or service applications; it may be preferable to use a custom **`ReportInfoSource`** in these situations. If your assembly does not already reference the appropriate Windows Forms libraries, references will be added.


ErrorReportDialog

ErrorReportDialog has encountered a problem and needs to close. We are sorry for the inconvenience.

Value does not fall within the expected range.

Please tell PreEmptive Solutions about this problem.

To help improve the software you use, PreEmptive Solutions is interested in learning more about this error. We have created a report about the error for you to send to us.

Read our online privacy policy.

Using a Custom ReportInfoSource

The ExceptionTrackAttribute defines three properties for specifying a **`ReportInfoSource`**:

- **ReportInfoSourceElement**. The **`ReportInfoSourceElement`** can be any of the values defined in the **`SourceElements`** enumeration: a field, a property, a method, or when using method-level exception tracking, a method argument. If the **`ReportInfoSourceElement`** is a method argument, it must correspond to a method parameter on the method to which the attribute is attached.
- **ReportInfoSourceOwner**. If the **`ReportInfoSourceElement`** is a field, method, or property, **`ReportInfoSourceOwner`** must indicate the class that defines the field, method, or property.
- **ReportInfoSourceName**. The **`ReportInfoSourceName`** should be set to the name of the field, method, property, or method argument of type **`IDictionary`** or **`IDictionary<string,string>`** that contains the user-specified report information at runtime.

These properties are described in detail in the ExceptionTrack section of the custom attribute reference.

There are three key-value pairs that may be included in the dictionary provided by the **`ReportInfoSource`**:

- **consent**. This is a string representation of a boolean that indicates whether the user has explicitly opted-in or out of sending the current exception report message. This consent is independent of and overrides the global PreEmptive Analytics opt-in setting.
- **comment**. This is a custom comment that is optionally provided by the user. It can be used to solicit feedback from the user, such as what he or she was doing when the exception occurred.
- **contact**. This is a contact point that is optionally provided by the user. Its content is not structured, and may for example contain an address, phone number, or user name for a social networking website. The built-in dialog requests the user provide this as an address.

Any key-value pairs other than those described above will be ignored.

Collecting user-specified report information is optional. If the retrieved dictionary is null, does not contain a `consent` key, or the value for the `consent` key is null or does not parse to a boolean, the global PreEmptive Analytics opt-in setting is respected. If the `comment` or `contact` keys are omitted, the resulting PreEmptive Analytics message does not include this information. See Entry Point Attributes for more information about the global PreEmptive Analytics opt-in setting.

Sample Exception Track attribute usage with ReportInfoSource defined as a method called "**`GetDictionary`**":

Exception Track Attribute Usage with ReportInfoSource

```
[ExceptionTrack(
   ReportInfoSourceElement = SourceElements.Method,
   ReportInfoSourceName = "GetDictionary"
)]
private void Foo() {
  ...
}

// Creates and populates a dictionary with user-specified report information
public IDictionary<string, string> GetDictionary() {
  Dictionary<string, string> dict = new Dictionary<string, string>();
  dict.Add("consent", "true");
  dict.Add("comment", "The Foo() method threw an exception.");
  dict.Add("contact", "foo@bar.com");
  return dict;
}
```

2.5.10.4 PreEmptive Analytics

PreEmptive Analytics Message Types

PreEmptive Analytics defines several message types:

- Application and Session Start
- Application and Session Stop
- Feature
- Performance Probe
- System Profile
- Tamper Detected
- Exception Detected

Application and Session **Start** and **Stop** messages (the *application lifecycle messages*) are intended to be sent when an *application* starts running and when it shuts down. The information contained in these messages is used to track application behavior and basic usage patterns. Extended usage and environment information is obtained by using the **Feature**, **Performance Probe**, or **System Profile** messages.

The data from these messages drive the Runtime Intelligence Portal’s dashboards. To have your application send these messages, you must:

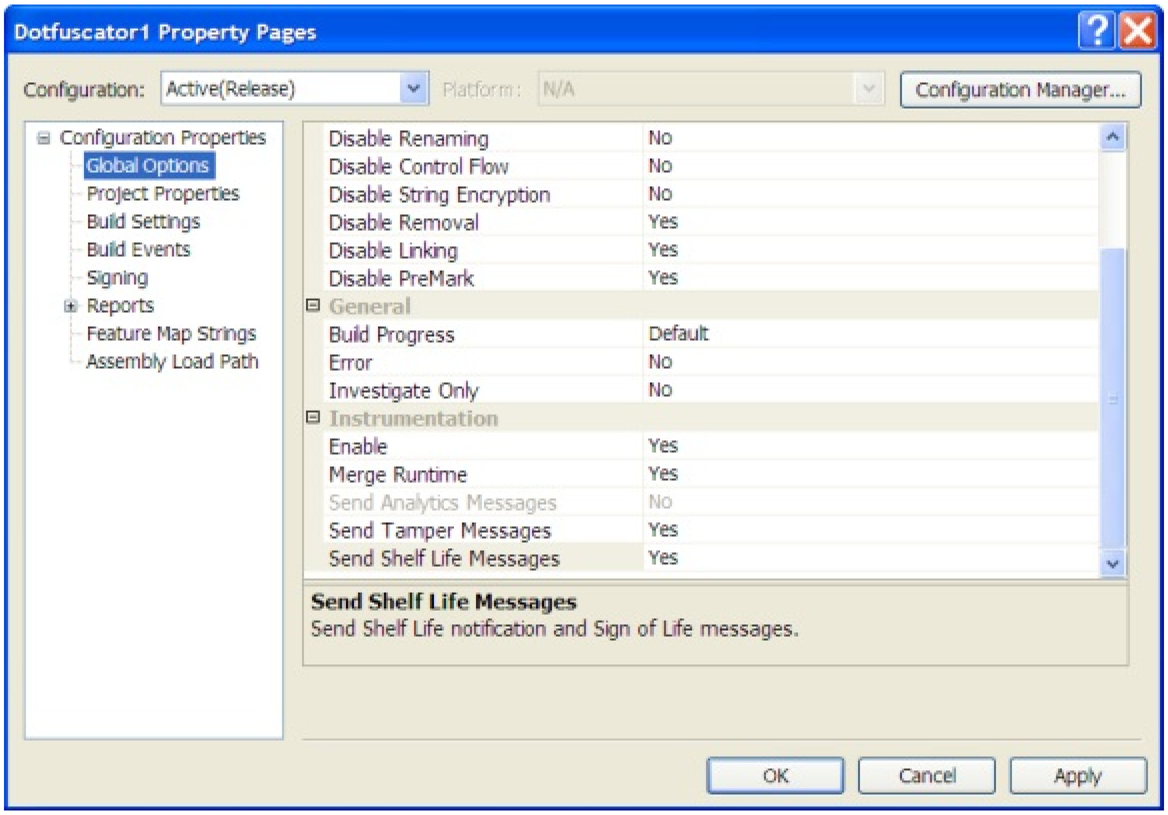
- Be a Runtime Intelligence Service subscriber (this gives you access to the dashboards and data in the portal).
- Instrument your application with PreEmptive Analytics attributes, including Setup and Teardown.
- Run your application through Dotfuscator with the **Send Analytics Messages** option turned on. See Configuring and Running Dotfuscator with Application Analytics.

See Example PreEmptive Analytics Enabled Application to see the contents of messages containing PreEmptive Analytics data.

2.5.10.4.1 Configuring and Running Dotfuscator with Application Analytics

PreEmptive Analytics Options

PreEmptive Analytics configuration options are available on the *Settings Tab* > *Global Options* property page in Dotfuscator. From within Visual Studio, the *Global Options* property page is available from the Dotfuscator project's properties dialog.



Disable Renaming	No
Disable Control Flow	No
Disable String Encryption	No
Disable Removal	Yes
Disable Linking	Yes
Disable PreMark	Yes
General	
Build Progress	Default
Error	No
Investigate Only	No
Instrumentation	
Enable	Yes
Merge Runtime	Yes
Send Analytics Messages	No
Send Tamper Messages	Yes
Send Shelf Life Messages	Yes

Send Shelf Life Messages
Send Shelf Life notification and Sign of Life messages.

From the standalone user interface, the *Options* property page is available on the **Settings** tab.

There are five options for Instrumentation (Tamper Detection, Shelf Life, Exception Reporting, and PreEmptive Analytics): **Enable**, **Merge Runtime**, **Send Analytics Messages**, **Send Shelf Life Messages**, and **Send Tamper Messages**.

Enabling Instrumentation (Tamper Detection, Shelf Life, and PreEmptive Analytics)

The **Enable** option turns Tamper Detection, Shelf Life, and PreEmptive Analytics on (default) and off. If this is turned off, Dotfuscator ignores all Tamper Detection, Shelf Life, and Application Analytics attributes contained within the input assemblies.

Merging or Referencing the Runtime

If set to "`Yes`" (default), the **Merge Runtime** option tells Dotfuscator to add the runtime code required for Tamper Detection, Shelf Life, and PreEmptive Analytics to one of the input assemblies. Distribution of the PreEmptive Analytics DLL is not required.

If set to "`No`", Dotfuscator outputs the runtime code in a separate assembly and adds the appropriate assembly references to the input assemblies. The new DLL must be distributed along with the application. Dotfuscator will still need to inject instrumentation helper methods into each assembly that is configured for Tamper Detection, Shelf Life, or PreEmptive Analytics.

Regardless of how the merge is set, Dotfuscator still needs to inject code into one of the input assemblies in order to use the PreEmptive Analytics library. Dotfuscator performs a dependency analysis of the input assemblies in order to choose the best one. It chooses in such a way as to minimize new dependencies among input assemblies; however, adding new dependencies is unavoidable in some cases. You can override the assembly that the PreEmptive Analytics runtime code will be injected into by specifying a Project Property with the name `accesspoint` and the value being the name of the assembly where the code will be inserted.

Strong Names

If any of the input assemblies are strong named, Dotfuscator will strong name the runtime DLL and sign it transparently. If none of the input assemblies are strong named, Dotfuscator will not strong name the runtime DLL.

Sending Tamper or Application Analytics Messages

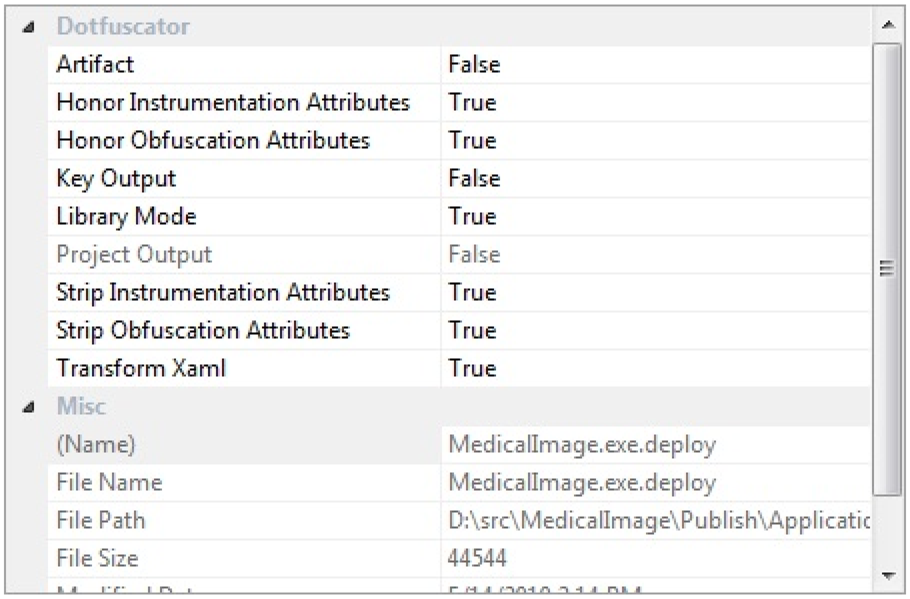
By default, assemblies that contain a method decorated with an **`InsertTamperCheck`** attribute are instrumented with code to send tamper notification messages when tampering is detected. This functionality is configurable and is controlled by the **Send Tamper Messages** option.

If the **Send Analytics Messages** option is set to "Yes", Dotfuscator will add code, based on the specified attributes, to send application start and stop messages, session start and stop messages, and feature usage messages.

Assembly Level Configuration Options

Dotfuscator allows granular control of instrumentation attribute handling (e.g. tamper detection, shelf life, and application analytics). The developer can tell Dotfuscator to honor or ignore and whether to keep or remove these attributes. These settings can be applied at the assembly level.

From within Visual Studio, assembly level options are available on the properties tool window when an input assembly has the focus.



Dotfuscator	
Artifact	False
Honor Instrumentation Attributes	True
Honor Obfuscation Attributes	True
Key Output	False
Library Mode	True
Project Output	False
Strip Instrumentation Attributes	True
Strip Obfuscation Attributes	True
Transform Xaml	True
Misc	
(Name)	MedicalImage.exe.deploy
File Name	MedicalImage.exe.deploy
File Path	D:\src\MedicalImage\Publish\Applicatic
File Size	44544

From the standalone user interface, the assembly level options are available on the *Input* tab.

Honor Instrumentation Attributes

Set *Honor Instrumentation Attributes* to "`True`" (the default) to tell Dotfuscator to process these attributes and perform the indicated instrumentation on the target assembly. Setting the property to "`False`" tells Dotfuscator to ignore any Instrumentation attributes.

A "`False`" setting is useful in testing scenarios and in advanced scenarios where a set of assemblies must be run through Dotfuscator multiple times.

Strip Instrumentation Attributes

Set *Instrumentation* attributes to "`True`" (the default) to tell Dotfuscator to remove these attributes from the target output assembly. Setting this property to "`False`" tells Dotfuscator to leave the attributes in the output assembly.

Like the *Honor Instrumentation Attributes* option, a "`False`" setting is useful in testing scenarios and in advanced scenarios where a set of assemblies must be run through Dotfuscator multiple times.

The table below indicates the results when combining these two options in different ways.

Honor Instrumentation Attributes	Strip Instrumentation Attributes	Notes
True	**True**	Default settings. The assembly is instrumented and attributes are removed.
False	**True**	The assembly is not instrumented and the attributes are stripped out. This is useful for creating test builds

		that do not include Tamper Detection or Application Analytics functionality.
True	**False**	The assembly is instrumented and the attributes are left in. This is currently not a recommended combination.
False	**False**	The assembly is not instrumented and the attributes are left in. This is useful for assemblies that need to be obfuscated but will need to have Tamper Detection or Application Analytics added in a subsequent step.

PreEmptive Analytics and other Dotfuscator Features

When using Application Analytics, other Dotfuscator features are available. The injected runtime code is annotated with obfuscation attributes so user configuration (beyond what is necessary for the input assemblies) is not required to perform renaming, removal, or other transforms on applications using Application Analytics.

2.5.10.4.2 PreEmptive Analytics Custom Attributes

All PreEmptive Analytics custom attributes are defined in `PreEmptive.Attributes.dll`, which is located by default in the *Dotfuscator 4* installation folder. To add PreEmptive Analytics custom attributes to an application, the developer must add a reference to this `DLL` and the `DLL` must be available at compile time. While injecting PreEmptive Analytics code, Dotfuscator removes references to this `DLL`; therefore, the `DLL` is not required at application runtime and does not need to be distributed with the application.

In addition to using the custom attributes `DLL`, all PreEmptive Analytics attributes may be specified as extended attributes using the instrumentation editor on the Dotfuscator user interface. This is useful if you do not want to modify the application source code to add custom attributes.

This section discusses the custom attributes at a high level (what they are, when and where to use them). For a programmer's reference, see the custom attribute reference.

2.5.10.4.2.1 Assembly Level Attributes

All PreEmptive Analytics messages need to contain common information about the entity and application sending the message. The developer provides this information in a set of custom attributes that are placed on one or more of the assemblies making up the application.

Unlike most other attributes, Dotfuscator does not remove assembly level attributes from the assembly after processing. Rather than removing them, Dotfuscator translates them into a form that the PreEmptive Analytics runtime can read when the application loads.

For more information, see the custom attribute reference.

Unique Identifiers

Most of the Assembly level attributes require a unique identifier. Except where noted, the identifiers must be generated and maintained by the developer. It is recommended that generated identifier strings use the standard GUID format. Examples:

Examples of Unique Identifiers
`123D35B3-BFDD-4797-9E9D-A39A57C1FD7B`
`B85A9684-1964-489e-867B-D81E89DB7CCB`

Application Attribute

Every assembly containing an entry point method should also have an Application attribute. The Application attribute provides information about the application sending PreEmptive Analytics data. Most important is the application ID, a unique identifier that should not change over the lifetime of the application. The application ID is required to allow a PreEmptive Analytics Endpoint to aggregate data about an application across name and version number changes.

The application attribute has other properties, including name and version. These properties are not required. If not specified in the application attribute, the PreEmptive Analytics code will try to acquire the application name and version by reflecting on the assembly. Alternatively, if the name and version are specified in the application attribute, then the PreEmptive Analytics code uses these values instead of reflecting on the assembly.

Binary Attribute

Every assembly containing a tamper checking method must also have a Binary attribute. The Binary attribute provides information about a specific component (assembly) in the application. Each assembly must have its own Binary attribute with its own corresponding ID. IDs specified in this attribute are sent in Tamper Notification messages to identify specific binaries (assemblies) that make up an application. This attribute is not required to be present.

Note: Binary information is not sent as part of application start and stop messages; it is sent as part of tamper detection messages.

Business Attribute

Every assembly containing an entry point method should also have a Business attribute. The Business attribute provides information about the organization that built the application. The most important property is the company key. The company key is a unique identifier that is provided as part of the PreEmptive Analytics enrollment process.

The Business attribute and company key are required for PreEmptive Analytics to work properly.

2.5.10.4.2.2 Entry Point Attributes

To send any kind of PreEmptive Analytics Data, identify appropriate entry point methods in the application to be processed. Then add Setup attributes to the entry point methods. When Dotfuscator encounters a Setup attribute during its processing, it adds PreEmptive Analytics initialization code to the start of the entry point method. If Dotfuscator is configured to *send analytics messages*, it adds code to send a startup message to a PreEmptive Analytics Endpoint. Setup attributes are not required at runtime; therefore, Dotfuscator strips them from the output application.

An entry point method is a method in the application that is executed as part of the application's startup sequence. It does not have to be the first method called by the runtime or framework when the application starts, but it must have the property that is executed every time the application is run.

Example entry point methods for different application types:

Application Type	Possible Entry Point Method
Console Application	Main or method always called from Main
Windows Forms Application	Main Form’s constructor

The Setup attribute has several arguments or properties, all in order to support privacy and data security scenarios.

Privacy and Security Settings

Opt-in

Applications that allow the user to opt-in or opt-out of PreEmptive Analytics Data gathering must communicate the user's choice to the PreEmptive Analytics code when the application starts.

To accomplish this, the Setup attribute is used to specify an `OptInSource`, which Dotfuscator uses to generate code that acquires the user's opt-in preference at runtime. The `OptInSource` is a Boolean-valued property, field, or method argument; it is the developer's responsibility to ensure that a correct value is stored in the `OptInSource` at the time the entry point method is executed.

The Setup attribute defines three properties for specifying an `OptInSource`:

- **OptInSourceElement**. The `OptInSourceElement` can be any of the values defined in the `SourceElements` enumeration: a field, a property, or a method argument. If the `OptInSourceElement` is a method argument, it must correspond to a method parameter on the method to which the `Setup` attribute is attached.
- **OptInSourceOwner**. If the `OptInSourceElement` is a field or property, `OptInSourceOwner` must indicate the class that defines the field or property.
- **OptInSourceName**. The `OptInSourceName` must be set to the name of the boolean field, property, or method argument that contains the user's opt-in preference at runtime.

The opt-in settings are optional. If they are omitted, the PreEmptive Analytics code sends all messages (*i.e.* it assumes opt-in).

Sample Setup attribute usage with `OptInSource` as method argument:

Sample Setup Attribute with OptInSource

```
[Setup(
   OptInSourceElement = SourceElements.MethodArgument,
   OptInSourceName = "optIn",
   UseSSL = false
)]
private void MyEntryMethod(bool optIn) { ... }
```

Application Instance ID

Applications that wish to provide an application instance identifier (e.g. a serial number) as part of the PreEmptive Analytics Data must communicate the ID to the PreEmptive Analytics code when the application starts.

To accomplish this, the Setup attribute can be used to specify an **`InstanceIdSource`**, which Dotfuscator uses to generate code that acquires the application's instance ID at runtime. The **`InstanceIdSource`** is a string valued property, field, or method argument; it is the developer's responsibility to ensure that a correct value is stored in the **`InstanceIdSource`** at the time the entry point method is executed.

Like the **`OptInSource`**, the **`Setup`** attribute defines three properties for specifying an **`InstanceIdSource`**:

- **InstanceIdSourceElement**. The **`InstanceIdSourceElement`** can be any of the values defined in the **`SourceElements`** enumeration: a field, a property, or a method argument. If the **`InstanceIdSourceElement`** is a method argument, it must correspond to a method parameter on the method to which the **`Setup`** attribute is attached.
- **InstanceIdSourceOwner**. If the **`InstanceIdSourceElement`** is a field or property, **`InstanceIdSourceOwner`** must indicate the class that defines the field or property.
- **InstanceIdSourceName**. The **`InstanceIdSourceName`** must be set to the name of the string field, property, or method argument that will contain the application's instance Id at runtime.

The instance ID settings are optional. If they are omitted, PreEmptive Analytics Data does not include an instance ID.

Sample Setup attribute usage with **`InstanceIdSource`** defined as a static property called "**`InstanceId`**" on the "**`Program`**" class:

Sample Setup Attribute with InstanceIdSource

```
[Setup(
   InstanceIdSourceElement = SourceElements.Property,
   InstanceIdSourceName = "InstanceId",
   InstanceIdSourceOwner = typeof(Program),
   UseSSL = false
)]
private void MyEntryMethod() { ... }
```

SSL

The UseSSL argument controls which web protocol the PreEmptive Analytics code uses to send messages. If UseSSL is set to true (default), it sends messages using the HTTPS protocol. If UseSSL is set to false, it sends messages using the HTTP protocol.

- The privacy and security settings properties are described in detail in the SetupAttribute section of the custom attribute reference.

Offline Storage of Usage Data

PreEmptive Analytics instrumented applications have the ability to store usage data in situations when network access is unavailable and then transmit the data when connectivity is restored. Usage data is stored in Isolated Storage. This behavior is enabled by default and default connectivity detection code is injected into instrumented applications. Developers can override the default behavior by changing the `OfflineStateSourceElement`. If the `OfflineStateSourceElement` value is changed to **None** then usage data will not be stored when the application is unable to connect to the network and that usage data will be dropped. Developers also have the ability to write their own network detection code and make the network connectivity state available to the PreEmptive Analytics code by specifying the applications offline state in a boolean value in the instrumented method's parameters, as the return value of a method or in a field or property. This is accomplished by setting the `OfflineStateSourceElement` property to the appropriate value and setting the `OfflineStateSourceName` and `OfflineStateSourceOwner`.

The application can also be notified of the success or failure of an attempt to store usage data to the offline storage mechanism via the `OfflineStorageResultSinkElement`. If the value is **None** then no notification will be made of the success or failure of the data storage. If the value is **DefaultAction** then if the storage mechanism is unable to store any usage data the application will exit immediately. Developers can write code to react to the success or failure of offline storage by setting the OfflineStorageResultSinkElement to the appropriate value and setting the `OfflineStorageResultSinkName` and `OfflineStorageResultSinkOwner`. The selected application code will be called with a parameter value or the boolean property or field will be set with the result of the most recent attempt to save usage data to the offline storage mechanism.

Custom Endpoint

When you add the SetupAttribute, you may set the destination of the messages using the StaticEndpoint property.

2.5.10.4.2.3 Exit Point Attributes

To prepare an application for analytics, the developer must identify appropriate exit methods. The developer then must add Teardown attributes to the exit methods. When Dotfuscator encounters a Teardown attribute during its processing, it adds analytics cleanup code to the end of the exit method. Teardown attributes are not required at runtime; therefore, Dotfuscator strips them from the output application after it adds the cleanup code.

An exit method is a method in the application that is executed as part of the application's shutdown sequence. It does not necessarily have to be the last method called, but it should have the property that it is executed exactly once when the application shuts down.

It is possible for the same method to be both an entry point and an exit method (for example, the Main method in a console application).

Example exit methods for different application types:

Application Type	**Possible Exit Method**
Console Application	Main or method always called from Main
Windows Forms Application	Main Form’s constructor
Windows Forms Application	Main Form's Dispose method
Windows Forms Application	Main Form's OnClosed event handler

The **`Teardown`** attribute has no required arguments or properties, so it is simple to use:

Teardown Attribute

```
[Teardown()]
private void MyExitMethod() { ... }
```

2.5.10.4.2.4 Tamper Notification Attributes

The Tamper Notification custom attribute is defined in **`PreEmptive.Attributes.dll`**, which is located by default in the *Dotfuscator 4* installation folder. To add the Tamper Notification custom attribute to an application, the developer must add a reference to this DLL and the DLL must be available at compile time. While injecting Tamper Notification code, Dotfuscator removes references to this DLL; therefore, the DLL is not required at application runtime and does not need to be distributed with the application.

In addition to using the custom attributes DLL, the Tamper Notification attribute may be specified as an extended attribute using the instrumentation editor on the Dotfuscator user interface. This is useful if you do not want to modify the application source code to add custom attributes.

This section discusses the custom attribute at a high level (what it is, when and where to use it). For a programmer's reference, see the custom attribute reference.

2.5.10.4.2.5 Shelf Life and Sign of Life Attributes

The Shelf Life and Sign of Life custom attributes are defined in **`PreEmptive.Attributes.dll`**, which is located by default in the *Dotfuscator 4* installation folder. To add the Shelf Life or Sign of Life custom attributes to an application, the developer must add a reference to this DLL and the DLL must be available at compile time. While injecting the Shelf Life code, Dotfuscator removes references to this DLL; therefore, the DLL is not required at application runtime and does not need to be distributed with the application.

In addition to using the custom attributes DLL, the Shelf Life or Sign of Life attributes may be specified as an extended attribute using the instrumentation editor on the Dotfuscator user interface. This is useful if you do not want to modify the application source code to add custom attributes.

This section discusses the custom attribute at a high level (what it is, when and where to use it). For a programmer's reference, see the custom attribute reference.

2.5.10.4.2.6 Exception Tracking Attributes

The Exception Track custom attribute is defined in **`PreEmptive.Attributes.dll`**, which is located by default in the *Dotfuscator 4* installation folder. To add the Exception Track custom attribute to an application, the developer must add a reference to this DLL and the DLL must be available at compile time. While injecting the Exception Tracking code, Dotfuscator removes references to this DLL; therefore, the DLL is not required at application runtime and does not need to be distributed with the application.

In addition to using the custom attributes DLL, the Exception Track attribute may be specified as an extended attribute using the instrumentation editor on the Dotfuscator user interface. This is useful if you do not want to modify the application source code to add custom attributes.

This section discusses the custom attribute at a high level (what it is, when and where to use it). For a programmer's reference, see the custom attribute reference.

2.5.10.4.2.7 Feature Usage Attributes

Dotfuscator provides support for feature usage tracking via the Feature attribute. The developer may add a Feature attribute to any method which maps to the start, stop, or entirety of a feature. When Dotfuscator encounters a Feature attribute during its processing, and it is configured to “send analytics messages”, it adds code to the method to send a PreEmptive Analytics feature usage message. Feature attributes are not required at runtime; therefore, Dotfuscator strips them from the output application.

The Feature attribute has several arguments or properties. Developers wishing to implement feature usage tracking with the PreEmptive Analytics Service will need to understand the use of the following properties:

Name

In order to make sense of feature-level analytics, features must be identified by a name. The Name argument is a string value which defines the name of the feature in question, and is required. This name need not follow any particular convention; but it should be descriptive and unique, except in cases where the feature attribute in question is one half of a start-stop pair in which case, the feature names should be identical.

Event Type

In order to identify what portion of the feature an attributed method implements, the **`FeatureEventType`** argument may be set. The **`FeatureEventType`** argument is an enumeration with the following values:

Enum Value	Implication
FeatureEventTypes.Tick	This method implements the entirety of the specified feature. (Default)
FeatureEventTypes.Start	This method's execution signifies the start of the specified feature. Code to send the message will be added to the start of the method.
FeatureEventTypes.Stop	This method's execution signifies the end of the specified feature. Code to send the message will be added to the end of the method.

The **`FeatureEventType`** argument is optional. If it is omitted, the default value (**`Tick`**) is assumed.

Sample Feature attribute usage for a method called as part of the **`Find`** feature:

Sample Feature Attribute

```
[Feature(
   "Find",
   FeatureEventType = FeatureEventTypes.Start
)]
private void BeginFind() { ... }
```

If a method's logic fully encompasses a feature, you may place two feature attributes on the method: a start and a stop. Dotfuscator sends the start message when the method begins execution, and the stop message when the method completes.

Start and Stop Feature Attributes

```
[Feature(
   "Find",
   FeatureEventType = FeatureEventTypes.Start
)]
[Feature(
   "Find",
   FeatureEventType = FeatureEventTypes.Stop
)]
private void BeginFind() { ... }
```

2.5.10.4.2.8 Performance Attributes

PreEmptive Analytics code can be used to gather and send performance related information while the application is executing. To add support for this to an application, place a **`PerformanceProbe`** attribute on a method or methods in the application. When Dotfuscator encounters the attribute during its processing, it adds code to obtain performance information and send a message to a PreEmptive Analytics Endpoint.

Performance data collected includes:

- CPU Utilization
- Memory available
- Memory used by current process

Performance Data

```
[PreEmptive.Attributes.PerformanceProbe()]
public void DoSomething() { ... }
```

The collected performance data is available in the *Data Extract* report on the Runtime Intelligence Portal. It can also be downloaded from the *File Feeds* section.

2.5.10.4.2.9 Environment Attributes

PreEmptive Analytics code can be used to gather and send information about the system the application is running on. To add support for this to an application, place a **`SystemProfile`** attribute on a method in the application. When Dotfuscator encounters the attribute during its processing, it adds code to gather the system profile and send a message to a PreEmptive Analytics Endpoint. Typically this data only needs to be collected once during an application run.

Below is a high level description of the kind of system data that is gathered:

Category	Examples of Collected Data
Processors	Number of processors, clock speeds, manufacturer, and processor ID.
Logical Disks	Number of logical disks, volume name, size, free space, file system
Memory	Speed, capacity
Network Adapters	IP address, MAC address
Domain	Domain name and role
Display	Name, refresh rate, vertical and horizontal resolution
Video	Name, memory size, color depth,
Terminal Services	Connections allowed
Sound	Name, manufacturer
Modem	Model, device type

Collected Data

```
[PreEmptive.Attributes.SystemProfile()]
public void Initialize() { ... }
```

The collected data is available in the *Data Extract* report on the Runtime Intelligence Portal. It can also be downloaded from the *File Feeds* section.

2.5.10.4.2.10 Sending User Defined Data with Extended Keys

Most PreEmptive Analytics message types allow user defined data (in the form of key-value pairs) to be gathered and sent along with the message.

To send extended key information, specify an **`ExtendedKeySource`** on the attribute corresponding to the message you wish to send.

Dotfuscator uses the **`ExtendedKeySource`** to generate code that gathers the key-value pairs at runtime. The **`ExtendedKeySource`** is an IDictionary or **`IDictionary<string,string>`** valued property, method, field, or method argument; it is the developer's responsibility to ensure that a correct value is available in the **`ExtendedKeySource`** at the time the attributed method is executed.

Attributes that support extended keys define three properties for specifying an **`ExtendedKeySource`**:

- **ExtendedKeySourceElement**. The **`ExtendedKeySourceElement`** can be any of the values defined in the **`SourceElements`** enumeration: a field, a property, a method, or a method argument. If the **`ExtendedKeySourceElement`** is a method argument, it must correspond to a method parameter on the method to which the attribute is attached.

- **ExtendedKeySourceOwner**. If the **`ExtendedKeySourceElement`** is a field, method, or property, **`ExtendedKeySourceOwner`** must indicate the class that defines the field, method, or property.
- **ExtendedKeySourceName**. The **`ExtendedKeySourceName`** should be set to the name of the IDictionary field, method, property, or method argument that contains the extended keys at runtime.

Extended key settings are always optional. If they are omitted, the resulting PreEmptive Analytics message does not include extended key information.

Sample Feature attribute usage with ExtendedKeySource defined as a method called "**`GetDictionary`**":

Feature Attribute Usage with ExtendedKeySource

```
[Feature(
..."Click"
   ExtendedKeySourceElement = SourceElements.Method,
   ExtendedKeySourceName = "GetDictionary",
)]
private void button1_Click(object sender, EventArgs e) {
...
}

// Creates and populates a dictionary with custom data
public IDictionary<string, string> GetDictionary() {
  Dictionary<string, string> dict = new Dictionary<string, string>();
  dict.Add("key1", "val1");
  dict.Add("key2", "val2");
  return dict;
}
```

Extended key data sent by the application is available in the Data Extract report on the Runtime Intelligence Portal. It can also be downloaded from the "*File Feeds*" section.

2.5.10.4.2.11 Automatically Sending Method Parameters as Extended Keys

In addition to user specified name-value pairs gathered at runtime and sent in the Extended Keys the parameter names and values of instrumented methods can be automatically gathered and added to the set of Extended Key data.

To add parameter information to extended key information, specify the **`ExtendedKeyMethodArguements`** on the attribute decorating the method whose parameters and values you wish to send.

Dotfuscator uses the **`ExtendedKeyMethodArguements`** to generate code that gathers the values of the specified parameters at runtime and places them in a key-value dictionary. The **`ExtendedKeyMethodArguements`** is a string that defines which parameters will be included, and optionally what the reported keys should named be in the message data.

Attributes that support extended keys provide an **`ExtendedKeyMethodArguement.`**

`ExtendedKeyMethodArguements` is always optional. If it is blank or omitted, the resulting PreEmptive Analytics message does not include parameter information.

`ExtendedKeyMethodArguements` values can consist of any combination of the following patterns:

*	All parameters
<param1>,<param2>	Only the names and values of param1 and param2
<param1>=<key1>,<param2>=<key2>	Only the values of param1 and param2 with the key names key1 and key2 respectively

2.5.10.4.3 Testing and Debugging Applications with Application Analytics

Once the application has been run through Dotfuscator, the Application Analytics functionality is ready to be tested. When the PreEmptive Analytics instrumented application is started and stopped, or when an attributed feature is used, the appropriate messages should be sent to the hosted service. There are several ways to verify that the correct messages are sent at the correct times. This section discusses client side and server side verification techniques.

2.5.10.4.3.1 Configuring Message Tracing

The developer can obtain a client side trace of outgoing messages by setting up message tracing in the application and examining the output as outlined below:

Here's a sample App.config File:

```
<configuration>
  <system.diagnostics>
    <trace autoflush="true" indentsize="0">
      <listeners>
        <remove name="Default"/>
        <add name="myListener" type="System.Diagnostics.TextWriterTraceListener"
initializeData="c:\myListener.log" />
      </listeners>
      </trace>
    <switches>
      <add name="traceSwitch" value="4" />
    </switches>
  </system.diagnostics>
</configuration>
```

The **`<listeners>`** element in the config file is where the developer can add and remove any or all listeners. Refer to the preceding example; it removes the default Trace Listener, which is **`DefaultTraceListener`** (the output window in Visual Studio), and adds the **`TextWriterTraceListener`**, which writes traces messages to c:\myListener.log.

TraceSwitch

This class provides support for multiple levels instead of the simple on/off control offered by the BooleanSwitch class. TraceSwitch class works with the following tracing levels:

Tracing Level	Configuration Value	Description
Off	**0**	Outputs no messages to Trace Listeners
Error	**1**	Outputs only error messages to Trace Listeners
Warning	**2**	Outputs error and warning messages to Trace Listeners
Info	**3**	Outputs informational, warning and error messages to Trace Listeners
Verbose	**4**	Outputs all messages to Trace Listeners

The name of the trace switch used in the message sending runtime is "**traceSwitch**", and the name of the config file trace switch name must be exactly the same for the tracing to work.

2.5.10.4.3.2 Watching Messages

HTTP traffic-logging tools such as Fiddler (available for download at: http://www.fiddlertool.com) allow the developer to watch PreEmptive Analytics messages on the wire, thereby providing another way to obtain the message ID.

It is easier to observe message traffic if SSL is turned off in the SetupAttribute.

2.5.10.4.3.3 Downloading Message Data

The Runtime Intelligence Portal provides the capability to securely download raw message data originating from .NET applications. The data is available in CSV files compatible with MS Excel.

To obtain raw message data, the developer should access the Runtime Intelligence Portal at http://www.runtimeintelligence.com. Enter the **User Name** and **Password** provided by PreEmptive Solutions, then navigate to *File Feeds*, located under *Data Extracts*.

2.5.10.4.4 Example PreEmptive Analytics Enabled Application

Here is a simple sample C# application scenario that uses custom attributes to drive instrumentation for application analytics.

Sample C# Scenario

```
[assembly: Business("66ce94c5-08c2-4b3b-99e4-0a92a5bb3c17", "PreEmptive Solutions")]
[assembly: Application("8F6A00E8-0C11-433e-A683-30EB828C4B3C")]
[assembly: Binary("444745EB-92CE-45e8-A749-33A0A92364FC")]

Entry point:

   static class Program
   {
      /// <summary>
      /// Demonstrating sample use of PreEmptive Analytics.
      /// </summary>
      [STAThread]
      static void Main() {
         SoSetup("357-1113-1719");
         Application.Run(new Form1());
      }

      [Setup(
          OptInSourceElement = SourceElements.None,
          InstanceIdSourceElement = SourceElements.MethodArgument,
          InstanceIdSourceName = "instanceId",
          UseSSL = false
       )]
      public static void SoSetup(string instanceId) {
      //empty method. Dotfuscator supplies the code (if the method is not
      // empty, Dotfuscator adds the code at the start of the method)
      }
   }

Exit point:

   public partial class Form1 : Form {
      [InsertTamperCheck()]
      public Form1() {
      // Dotfuscator will add application integrity checks here
      // along with code to send a message if the integrity check fails.
         InitializeComponent();
      }

      [Teardown]
      private void Form1_FormClosing(object s, FormClosingEventArgs e) {
      //empty method. Dotfuscator supplies the code (if the method is not
      // empty, Dotfuscator adds the code at the end of the method)
      }
   }
```

```
Application Startup message for SOSWinApp.exe, whose serial number is 357-1113-1719:

<MessageCache>
  <InstanceId>357-1113-1719</InstanceId>
  <ApplicationGroupId>1cb4c0ba-f9b1-4935-a633-7cdc2ff6d38b</ApplicationGroupId>
  <Business>
    <CompanyName>PreEmptive Solutions</CompanyName>
    <CompanyId>66ce94c5-08c2-4b3b-99e4-0a92a5bb3c17</CompanyId>
  </Business>
  <TimeSentUtc>2007-10-01T21:46:00.2250193Z</TimeSentUtc>
  <ApiLanguage>.NET CLR</ApiLanguage>
  <ApiVersion>2.0.2795.27414</ApiVersion>
  <Id>1a97253b-045b-486d-a097-c8ca5eea0efd</Id>
  <Messages>
    <Message xsi:type="ApplicationLifeCycle">
      <SessionId>1cb4c0ba-f9b1-4935-a633-7cdc2ff6d38b</SessionId>
      <Event>
        <PrivacySetting>SupportOptout</PrivacySetting>
        <Code>Application.Start</Code>
      </Event>
      <Binary>
        <ModifiedDate>2007-10-01T17:40:53.947891-04:00</ModifiedDate>
        <Id>444745eb-92ce-45e8-a749-33a0a92364fc</Id>
        <Name>SOSWinApp, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null</Name>
        <Version>1.0.0.0</Version>
      </Binary>
      <TimeStampUtc>2007-10-01T21:45:50.2500433Z</TimeStampUtc>
      <Id>63d588c4-761a-4f6a-90fe-f3d89432a797</Id>
      <User>
        <IsAdministrator>false</IsAdministrator>
        <Name>724908d2f82cafe6b03e51438dcc5838</Name>
      </User>
      <Host>
        <RuntimeVersion>2.0.50727.42</RuntimeVersion>
        <IPAddress>172.16.7.42</IPAddress>
        <Name>04f07368bbabe92b7494f5fdd72c6476</Name>
        <OS>
          <OsInstallDate>2006-10-24T13:45:38</OsInstallDate>
          <OsName>Microsoft Windows XP Professional</OsName>
          <OsServicePackMajorVersion>2</OsServicePackMajorVersion>
          <OsServicePackMinorVersion>0</OsServicePackMinorVersion>
          <Locale>0409</Locale>
          <OSLanguage>1033</OSLanguage>
          <IsVirtualized>false</IsVirtualized>
        </OS>
      </Host>
    </Message>
    <Message xsi:type="SessionLifeCycle">
      <SessionId>1cb4c0ba-f9b1-4935-a633-7cdc2ff6d38b</SessionId>
      <Event>
        <PrivacySetting>SupportOptout</PrivacySetting>
        <Code>Session.Start</Code>
      </Event>
      <Binary>
        <ModifiedDate>2007-10-01T17:40:53.947891-04:00</ModifiedDate>
        <Id>444745eb-92ce-45e8-a749-33a0a92364fc</Id>
        <Name>SOSWinApp, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null</Name>
        <Version>1.0.0.0</Version>
      </Binary>
      <TimeStampUtc>2007-10-01T21:45:50.4214882Z</TimeStampUtc>
      <Id>0e7c2426-dd62-44cd-ad66-4d9c05097af5</Id>
    </Message>
    <Message xsi:type="SessionLifeCycle">
      <SessionId>1cb4c0ba-f9b1-4935-a633-7cdc2ff6d38b</SessionId>
      <Event>
        <PrivacySetting>SupportOptout</PrivacySetting>
        <Code>Session.Start</Code>
      </Event>
      <Binary>
        <ModifiedDate>2007-10-01T17:40:53.947891-04:00</ModifiedDate>
        <Id>444745eb-92ce-45e8-a749-33a0a92364fc</Id>
        <Name>SOSWinApp, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null</Name>
        <Version>1.0.0.0</Version>
      </Binary>
      <TimeStampUtc>2007-10-01T21:45:50.4214882Z</TimeStampUtc>
      <Id>a44c5dd3-6328-47aa-82a2-63a28bf8270d</Id>
    </Message>
  </Messages>
  <SchemaVersion>02.00.00</SchemaVersion>
  <Application>
    <ApplicationType>.exe</ApplicationType>
    <Id>8f6a00e8-0c11-433e-a683-30eb828c4b3c</Id>
    <Name>SOSWinApp</Name>
    <Version>1.0.0.0</Version>
  </Application>
</MessageCache>

Application Shutdown for SOSWinApp.exe, whose serial number is 357-1113-1719:

<MessageCache>
  <InstanceId>357-1113-1719</InstanceId>
  <ApplicationGroupId>1cb4c0ba-f9b1-4935-a633-7cdc2ff6d38b</ApplicationGroupId>
```

```
<Business>
  <CompanyName>PreEmptive Solutions</CompanyName>
  <CompanyId>66ce94c5-08c2-4b3b-99e4-0a92a5bb3c17</CompanyId>
</Business>
<TimeSentUtc>2007-10-01T21:47:17.6401846Z</TimeSentUtc>
<ApiLanguage>.NET CLR</ApiLanguage>
<ApiVersion>2.0.2795.27414</ApiVersion>
<Id>2ab3e80c-2a37-4cbb-ae0c-905ba7f20f4d</Id>
<Messages>
  <Message xsi:type="SessionLifeCycle">
    <SessionId>1cb4c0ba-f9b1-4935-a633-7cdc2ff6d38b</SessionId>
    <Event>
      <PrivacySetting>SupportOptout</PrivacySetting>
      <Code>Session.Stop</Code>
    </Event>
    <Binary>
      <ModifiedDate>2007-10-01T17:40:53.947891-04:00</ModifiedDate>
      <Id>444745eb-92ce-45e8-a749-33a0a92364fc</Id>
      <Name>SOSWinApp, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null</Name>
      <Version>1.0.0.0</Version>
    </Binary>
    <TimeStampUtc>2007-10-01T21:47:17.5934269Z</TimeStampUtc>
    <Id>ae2d572d-6d22-4176-aa98-4ea69ef649fb</Id>
  </Message>
  <Message xsi:type="ApplicationLifeCycle">
    <SessionId>1cb4c0ba-f9b1-4935-a633-7cdc2ff6d38b</SessionId>
    <Event>
      <PrivacySetting>SupportOptout</PrivacySetting>
      <Code>Application.Stop</Code>
    </Event>
    <Binary>
      <ModifiedDate>2007-10-01T17:40:53.947891-04:00</ModifiedDate>
      <Id>444745eb-92ce-45e8-a749-33a0a92364fc</Id>
      <Name>SOSWinApp, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null</Name>
      <Version>1.0.0.0</Version>
    </Binary>
    <TimeStampUtc>2007-10-01T21:47:17.5934269Z</TimeStampUtc>
    <Id>086289bc-d8cb-453e-b5ea-b4ac238a4647</Id>
    <User>
      <IsAdministrator>false</IsAdministrator>
      <Name>724908d2f82cafe6b03e51438dcc5838</Name>
    </User>
    <Host>
      <RuntimeVersion>2.0.50727.42</RuntimeVersion>
      <IPAddress>172.16.7.42</IPAddress>
      <Name>04f07368bbabe92b7494f5fdd72c6476</Name>
```

```
        <OS>
          <OsInstallDate>2006-10-24T13:45:38</OsInstallDate>
          <OsName>Microsoft Windows XP Professional</OsName>
          <OsServicePackMajorVersion>2</OsServicePackMajorVersion>
          <OsServicePackMinorVersion>0</OsServicePackMinorVersion>
          <Locale>0409</Locale>
          <OSLanguage>1033</OSLanguage>
          <IsVirtualized>false</IsVirtualized>
        </OS>
      </Host>
    </Message>
  </Messages>
  <SchemaVersion>02.00.00</SchemaVersion>
  <Application>
    <ApplicationType>.exe</ApplicationType>
    <Id>8f6a00e8-0c11-433e-a683-30eb828c4b3c</Id>
    <Name>SOSWinApp</Name>
    <Version>1.0.0.0</Version>
  </Application>
</MessageCache>

Tamper detected message for SOSWinApp.exe, whose serial number is 357-1113-1719:

<MessageCache>
  <InstanceId>357-1113-1719</InstanceId>
  <ApplicationGroupId>b38041f6-f01c-4a2a-8ce5-b6caaa7d39e8</ApplicationGroupId>
  <Business>
    <CompanyName>PreEmptive Solutions</CompanyName>
    <CompanyId>66ce94c5-08c2-4b3b-99e4-0a92a5bb3c17</CompanyId>
  </Business>
  <TimeSentUtc>2007-10-01T21:51:12.5359398Z</TimeSentUtc>
  <ApiLanguage>.NET CLR</ApiLanguage>
  <ApiVersion>2.0.2795.27414</ApiVersion>
  <Id>47da5317-4df6-4f15-85cd-f0a01ed5c00c</Id>
  <Messages>
    <Message xsi:type="SecurityMessage">
      <SessionId>b38041f6-f01c-4a2a-8ce5-b6caaa7d39e8</SessionId>
      <Event>
        <PrivacySetting>AlwaysSend</PrivacySetting>
        <Code>Security.Integrity.Tampering</Code>
      </Event>
      <Binary>
        <ModifiedDate>2007-10-01T17:50:18.4525198-04:00</ModifiedDate>
        <Id>444745eb-92ce-45e8-a749-33a0a92364fc</Id>
        <Name>SOSWinApp, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null</Name>
        <Version>1.0.0.0</Version>
      </Binary>
      <TimeStampUtc>2007-10-01T21:51:02.7323458Z</TimeStampUtc>
      <Id>8f3c939a-79f3-4fbc-97b0-d8af5ef9ad4b</Id>
    </Message>
  </Messages>
  <SchemaVersion>02.00.00</SchemaVersion>
  <Application>
    <ApplicationType>.exe</ApplicationType>
    <Id>8f6a00e8-0c11-433e-a683-30eb828c4b3c</Id>
    <Name>SOSWinApp</Name>
    <Version>1.0.0.0</Version>
  </Application>
</MessageCache>
```

2.5.11 Decoding Obfuscated Stack Traces

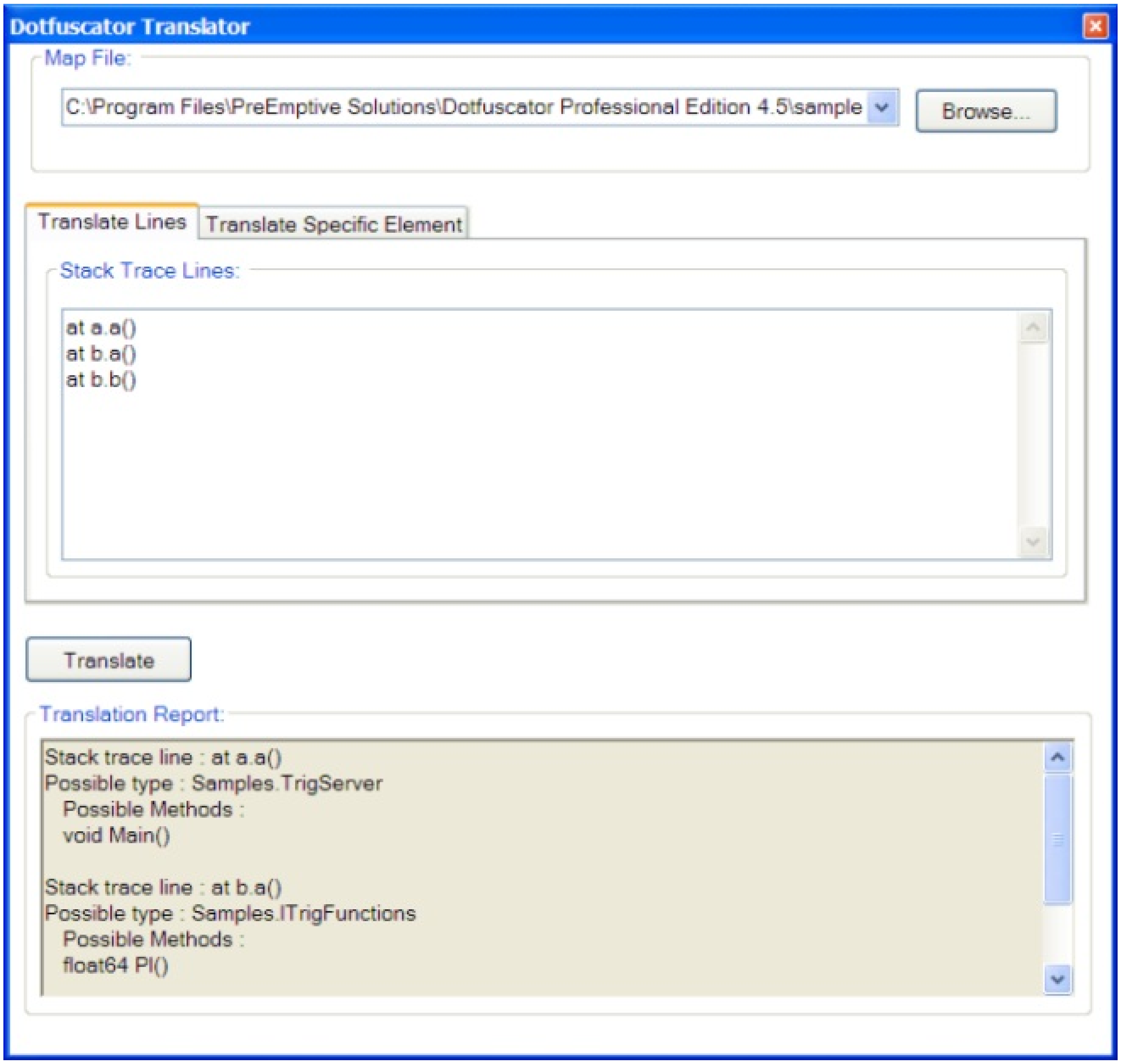
One potential drawback of obfuscation is that debugging and troubleshooting obfuscated applications can be difficult due to name mangling. Dotfuscator addresses this drawback by providing an integrated tool that allows you to use your output mapping files to recover the original symbols from obfuscated stack traces.

For example, if you have an obfuscated application that you have shipped and you receive a stack trace from one of your customers, that stack trace might look like this:

Stack Trace

```
Unhandled Exception: System.ApplicationException: A bad thing happened!
   at a.a()
   at b.a(String A_0)
   at b.a(String[] A_0)
```

You could use your XML mapping file, or better yet, the HTML Report based on the mapping file to manually recover the original names, but this can be a tedious and time consuming process. The stack trace translation tool automates this by letting you provide a map file, paste the stack trace into a window, and press the Translate button. The translated stack trace is shown at the bottom window:



C:\Program Files\PreEmptive Solutions\Dotfuscator Professional Edition 4.5\sample

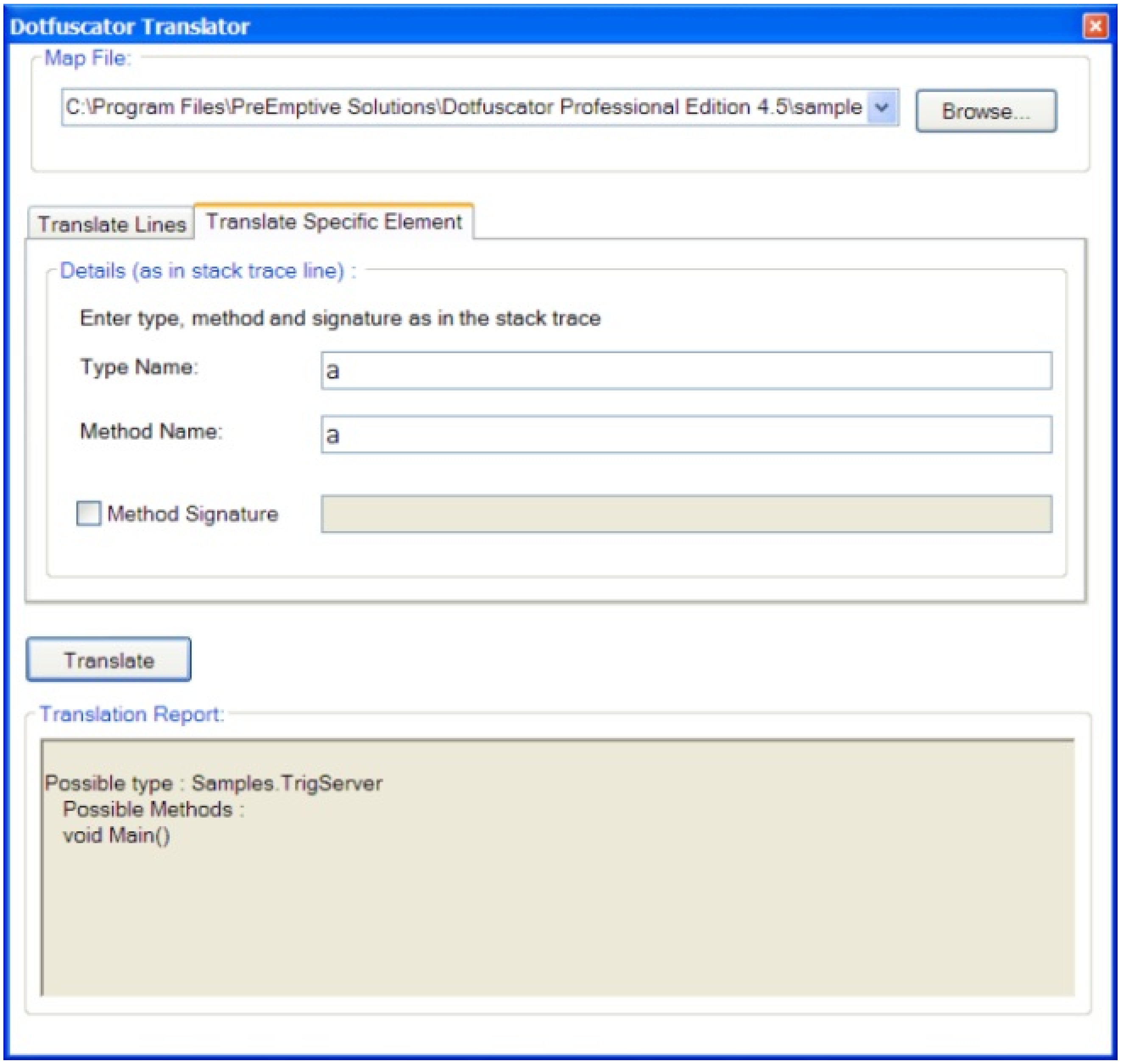
```
at a.a()
at b.a()
at b.b()
```

```
Stack trace line : at a.a()
Possible type : Samples.TrigServer
  Possible Methods :
  void Main()

Stack trace line : at b.a()
Possible type : Samples.ITrigFunctions
  Possible Methods :
  float64 PI()
```

Some methods in the obfuscated stack trace might be ambiguous; that is, due to the use of Overload Induction and Enhanced Overload Induction, there might be more than one matching un-obfuscated method. In these cases, the tool displays all the possibilities.

If you just want to look up a specific type or method by name, click the *Translate Specific Element* tab. You will see a screen that will allow you to type in the obfuscated names of the specific items you want to translate.


Dotfuscator Translator
Map File:
C:\Program Files\PreEmptive Solutions\Dotfuscator Professional Edition 4.5\sample
Browse...
Translate Lines
Translate Specific Element
Details (as in stack trace line) :
Enter type, method and signature as in the stack trace
Type Name:
Method Name:
Method Signature
Translate
Translation Report:

Type Name: a

Method Name: a

```
Possible type : Samples.TrigServer
  Possible Methods :
  void Main()
```

- To look up just a type, fill in the *Type Name* text box and click **Translate**.
- To look up a method, fill in the *Type Name* and *Method Name* text boxes. You can optionally provide a signature by checking the **Method Signature** check box and filling in the signature in the adjacent text box. The signature must be as it would appear in a stack trace.

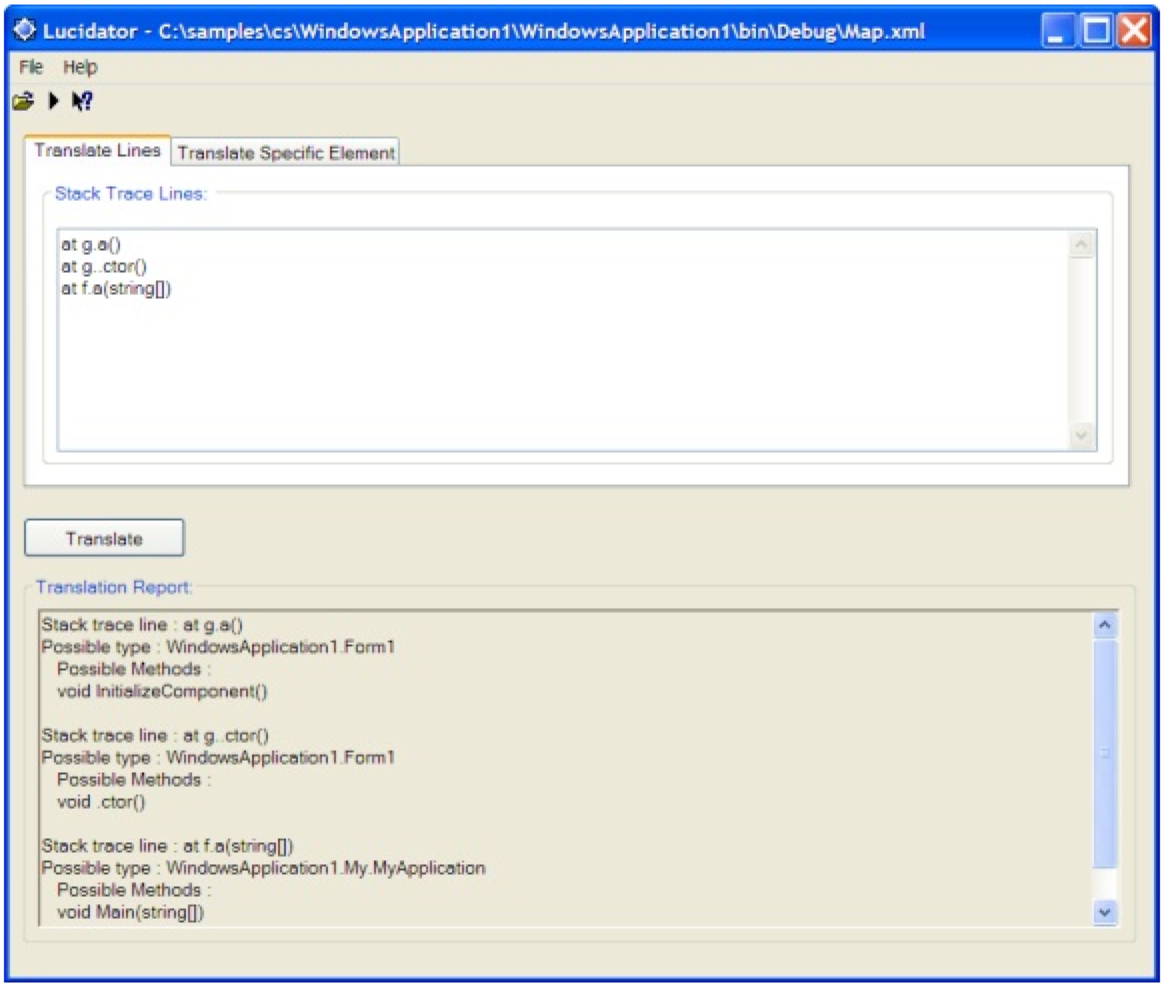
2.5.12 Using Lucidator

To translate complete stack traces, open a map file produced by Dotfuscator. Next, open Lucidator by clicking **Start > PreEmptive Solutions > Lucidator** or using the following commands in the command line:

Options	Description
`/mapfile=<map file>`	specifies the map file (eg: /mapfile=map.xml)
`/stacktracefile=<stack trace file>`	specifies the file containing the stack trace (eg: /stacktracefile=stacktrace.txt)
`/c=<culture>`	set user interface language (requires appropriate language resources). Argument is the lowercase language code:(eg: /c=de, /c=ja, /c=zh-CHS

When the Lucidator window displays on your desktop, select the *Translate Lines* tab, and then paste the stack trace in the window. Click the **Translate** button.

The translated stack trace displays in the *Translation Report* section of the window.


Lucidator - C:\samples\cs\WindowsApplication1\WindowsApplication1\bin\Debug\Map.xml
File
Help
Translate Lines
Translate Specific Element
Stack Trace Lines:
Translate
Translation Report:

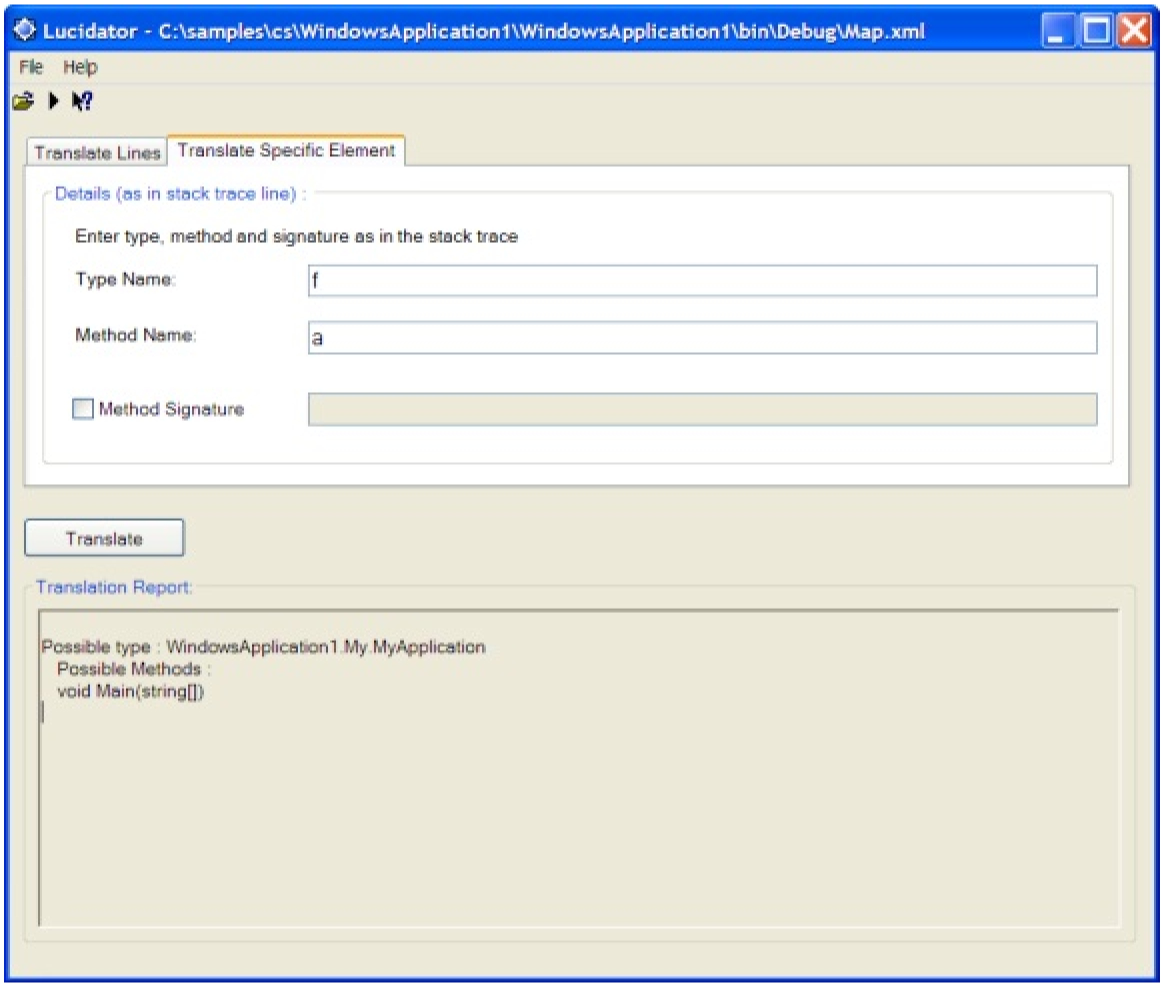
```
at g.a()
at g..ctor()
at f.a(string[])
```

```
Stack trace line : at g.a()
Possible type : WindowsApplication1.Form1
  Possible Methods :
  void InitializeComponent()

Stack trace line : at g..ctor()
Possible type : WindowsApplication1.Form1
  Possible Methods :
  void .ctor()

Stack trace line : at f.a(string[])
Possible type : WindowsApplication1.My.MyApplication
  Possible Methods :
  void Main(string[])
```

Methods in obfuscated stack traces may be ambiguous due to the use of Overload Induction and Enhanced Overload Induction. Because there may be more than one matching unobfuscated method, Lucidator displays all the possibilities. To look up a specific type or method by name, click the *Translate Specific Element* tab:


Lucidator - C:\samples\cs\WindowsApplication1\WindowsApplication1\bin\Debug\Map.xml
File Help
Translate Lines
Translate Specific Element
Details (as in stack trace line) :
Translate
Translation Report:

Enter type, method and signature as in the stack trace

Type Name: f

Method Name: a

Method Signature

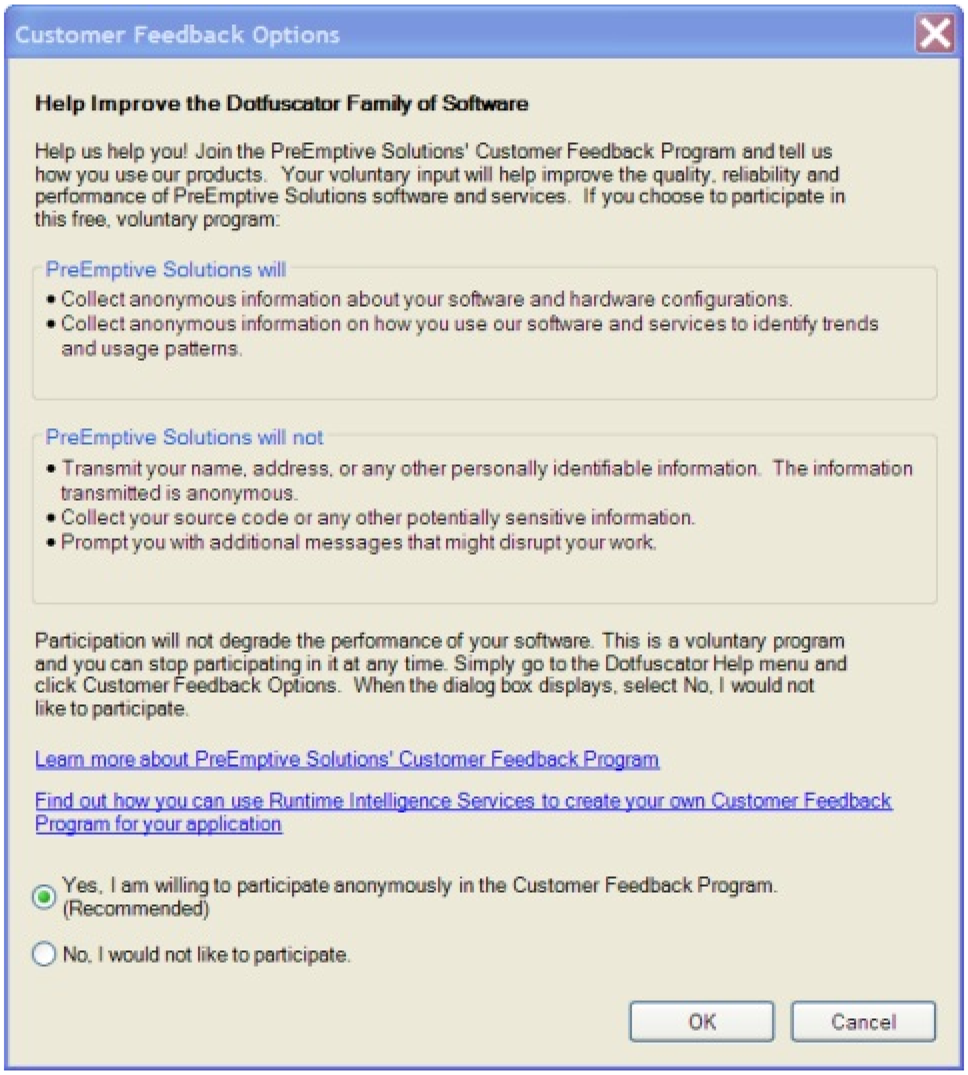
```
Possible type : WindowsApplication1.My.MyApplication
  Possible Methods :
  void Main(string[])
```

In this screen, enter the obfuscated names of the specific items you want to translate. For example, to translate an obfuscated type name, enter the obfuscated name in the **Type Name:** field and click **Translate**. Likewise, to translate an obfuscated method name, enter the obfuscated name in the **Method Name:** field and click **Translate**. You may optionally provide a signature a by checking the **Method Signature** box and entering the signature in the adjacent text box.

Note: The signature must be as it appears in the stack trace.

2.5.13 Customer Feedback Options

Dotfuscator provides an anonymous usage reporting system that users can opt-in to. If you opt-in to this program, only anonymized high level usage data will be gathered by PreEmptive Solutions with the sole intent of improving the Dotfuscator Software product family. You may change your feedback options at any time from the **Help > Customer Feedback Options** menu.



Help Improve the Dotfuscator Family of Software

Help us help you! Join the PreEmptive Solutions' Customer Feedback Program and tell us how you use our products. Your voluntary input will help improve the quality, reliability and performance of PreEmptive Solutions software and services. If you choose to participate in this free, voluntary program:

PreEmptive Solutions will
- Collect anonymous information about your software and hardware configurations.
- Collect anonymous information on how you use our software and services to identify trends and usage patterns.

PreEmptive Solutions will not
- Transmit your name, address, or any other personally identifiable information. The information transmitted is anonymous.
- Collect your source code or any other potentially sensitive information.
- Prompt you with additional messages that might disrupt your work.

Participation will not degrade the performance of your software. This is a voluntary program and you can stop participating in it at any time. Simply go to the Dotfuscator Help menu and click Customer Feedback Options. When the dialog box displays, select No, I would not like to participate.

Learn more about PreEmptive Solutions' Customer Feedback Program

Find out how you can use Runtime Intelligence Services to create your own Customer Feedback Program for your application

● Yes, I am willing to participate anonymously in the Customer Feedback Program. (Recommended)

○ No, I would not like to participate.

2.6 References

This section contains a full reference to all of Dotfuscator's command line syntax, build task settings, configuration file options, and custom attributes. It also contains pointers to Dotfuscator's configuration and mapping file DTDs.

Additional resources such as articles, whitepapers, and tutorials can be found online at www.preemptive.com/resources.html.

2.6.1 Command Line Interface Reference

The command line interface is designed to allow you to:

- Obfuscate from the command line without requiring creation of a configuration file.
- Override or supplement options in an existing configuration file using command line options.
- Create a configuration file from the command line.
- Launch the standalone graphical user interface with options and/or a configuration file specified on the command line.

2.6.1.1 Command Line Option Summary

Command line options may begin with the '**/**' or the '**-**' characters.

Command Line Options
`Usage: dotfuscator [options] [config_file]`

Traditional Options

The following is a summary of the traditional command line options.

Traditional Options	**Description**
`/g`	Launch the standalone GUI
`/i`	Investigate only
`/p=<property list>`	Specifies values for user defined properties in the configuration file. Comma separated list of name-value pairs (e.g. /p=projectdir=c:\\temp,projectname=MyApp.exe)
`/q`	Quiet output
`/v`	Verbose output
`/nologo`	Suppresses the output of the Dotfuscator Copyright and License information.
`/?`	Print help
`[config_file]`	configuration file containing runtime options.

The **`/v`** option induces Dotfuscator to provide information about its progress during execution. The level of detail here will likely change between releases.

The **`/i`** option tells Dotfuscator to not create any output assemblies files. If the configuration file specifies a map file, the results of the run will be found there (this option is close to worthless without generating a map).

The **`/q`** option tells Dotfuscator to run completely without printed output. This is suitable for inclusion into application build sequences. This option overrides verbose mode.

The **`/p`** option tells Dotfuscator to set external properties at the command line. Setting these properties here will override those specified in the **`<properties>`** section of the configuration file.

The <**proplist**> is a list of comma-separated name-value pairs. For example, property declaration and assignment in conjunction with the -**p** option, might look like:

Property Declaration and Assignment in Conjunction with -p Option:

```
/p=projectdir=c:\temp,projectname=MyApp
```

Properties may be quoted if they contain spaces as illustrated below:

Quoting Properties

```
/p=MyProperty="value has spaces"
```

Note: Property names are case sensitive.

The /**g** option tells Dotfuscator to start up the standalone graphical user interface. You can start the graphical user interface with external properties and a specific configuration file using this option:

Start Up Standalone GUI

```
Dotfuscator /g /p=projectdir=c:\temp project_template.xml
```

The graphical user interface starts up if Dotfuscator is run with no command line arguments.

The **configfile** is a configuration file that is required for every run of Dotfuscator. Notice you do not enter configuration information or target assemblies on the command line. This information must be found in the configuration file.

Extended Options

Extended options are designed to allow for basic obfuscation from the command line, without requiring you to first create a configuration file. If you use a configuration file with an extended command line option, the command line option supplements or overrides the commands in the configuration file. See Supplementing or Overriding a Configuration File from the Command Line for more information.

Extended options are recognized by the first four characters.

The following is a summary of the extended command line options. An asterisk denotes the default setting if an option is missing and no configuration file is specified.

Extended Options	Description
`/in [+|-]<file>[,[+|-]<file>]`	Specify inputs. Any combination of assemblies, package files or directory file masks can be specified. Use prefix to obfuscate input as library mode (**+**) or private (**-**) assembly. Default is governed by assembly file extension (EXEs are private; .DLLs are run in library mode).
`/out:<directory>`	Specify output directory. Default is **.\Dotfuscated**.

`/honor:[on|off*]`	Toggle honoring obfuscation attribute directives found in all input assemblies.
`/strip:[on|off*]`	Toggle stripping obfuscation attributes from all input assemblies.
`/makeconfig:<file>`	Save all runtime options (from command line and configuration file if present) to `<file>`.
`/debug:[on:off*|impl|opt|pdb]`	Emit debugging symbols for obfuscated assemblies and control JIT behavior.
`/suppress:[on|off*]`	Add the SuppressIldasmAttribute to supported output assemblies.
`/disable`	Disable all transforms regardless of other options
`/rename:[on|off*]`	Enable/disable renaming.
`/mapout:<file>`	Specify output mapping file. Default is **.\Dotfuscated\map.xml**.
`/mapin:<file>`	Specify input mapping file.
`/clobbermap:[on|off*]`	Specify map file overwrite mode.
`/keep:[namespace|hierarchy|none*]`	Specify type renaming scheme.
`/prefix:[on:off*]`	Append a prefix to all renamed types.
`/enhancedOI:[on|off*]`	Use Enhanced Overload Induction.
`/refsrename:[on*|off]`	Rename referenced metadata defined only in input map file.

`/naming:`
`[loweralpha*|upperalpha|numeric|unprintable]`
	Specify identifier renaming scheme.

`/controlflow:[high*|medium|low|off]`	Set control flow obfuscation level.
`/encrypt:[on*|off]`	Enable/disable string encryption.
`/prune:[on*|off|const]`	Enable/disable pruning, or enable constant-only pruning.
`/link:[[+]<name>[,[+]<name>],]out=<name>`	Specify assemblies to link into named output assembly. A '**+**' prefix indicates a prime assembly. Omit the list to link all inputs, using first input as the prime assembly. You can specify multiple link options on the command line.
`/link:off`	Disables linking. Linking is off by default unless you pass a configuration file with linking options. This option is useful for that scenario.
`/premark:[on|off*|only]`	Enable/disable watermarking. "`Only`" option disables all other

	transforms.
`/watermark:<string>`	Specify the watermark string. Quotes are optional. By default, all input assemblies will be watermarked with this string.
`/passphrase:<passphrase>`	Optionally specify a passphrase to use for encrypting the watermark string.
`/charmap:<name>`	Specify a character map to use to encode the watermark string. The name must be one of the supported character maps. See Character Maps.
`/smart:[on|off]`	Enables/disables Smart Obfuscation. See SmartObfuscation.
`/soreport:[all|warn|none]`	Sets the verbosity level of the reporting output of the Smart Obfuscation functionality. See SmartObfuscation.
`/offlineactivation:<activation_file>`	Manually activate Dotfuscator with the activation data contained in the <activation_file>.

Examples:

Example 1

```
dotfuscator -in:my.dll
```

Obfuscates **`my.dll`** as a library (visible symbols preserved and unpruned) with renaming, control flow, pruning, and string encryption turned on. The output assembly is written to a directory called **.\Dotfuscated**, and the map file is written to **.\Dotfuscated\map.xml** since no output directories were specified.

Example 2

```
dotfuscator -in:-myapp.exe,-private.dll
```

Obfuscates **`myapp.exe`** and **`private.dll`** together as a standalone application. Even visible symbols inside the DLL are obfuscated. Pruning is enabled based on the entry point method contained in myapp.exe.

Example 3

```
dotfuscator -in:myapp.exe -mapo:MyName.xml
```

This command obfuscates **`myapp.exe`** as a standalone application. An output renaming map is specified.

2.6.1.2 Supplementing or Overriding a Configuration File from the Command Line

Dotfuscator has the unique ability to accept a complete or partial configuration file, yet allow you to supplement or override its options from the command line. This allows you to quickly adjust and tweak settings using a standard configuration file as a template.

Command Line Option	**Configuration File Option**	**Notes**
`/in [+|-]<file>[,[+|-]<file>]`	*input* section	adds
`/out: <directory>`	*output* section	overrides
`/honor:[on|off*]`	*inputassembly* section	overrides
`/strip:[on|off*]`	*inputassembly* section	overrides
`/debug:[on|off*|impl|opt|pdb]`	"`debug`" global option	overrides
`/suppress:[on|off*]`	"`suppressildasmattribute`" global option	overrides
`/disable`	Sets "`disable`" option in *renaming, controlflow, stringencrypt,* and *removal* sections	overrides
`/rename:[on:off]`	Sets (or unsets) "`disable`" option in "*renaming*" section.	Overrides

`/mapout:<file>`	"*mapoutput*" section	overrides
`/mapin:<file>`	"*mapinput*" section	overrides
`/clobbermap:[on|off]`	"`overwrite`" attribute in "*mapoutput*" section	overrides
`/keep:[namespace|hierarchy|none]`	Sets (or unsets) renaming options: "`keepnamespace`", "`keephierarchy`"	overrides
`/enhancedOI:[on|off]`	Sets (or unsets) "`enhancedOI`" renaming option	overrides
`/refsrename:[on|off]`	"`obfuscatereferences`" attribute in "`mapinput`" element.	overrides
`/naming:[loweralpha|upperalpha|numeric|unprintable]`	Sets the "`scheme`" attribute in the *renaming* section.	overrides
`/controlflow:[high|medium|low|off]`	Sets the "`level`" attribute in the *controlflow*" section. The `off` flag sets the "`disable`" option.	overrides
`/encrypt:[on|off]`	Sets (or unsets) the "`disable`" option in the *stringencrypt* section.	overrides

`/prune:[on|off]`	Sets (or unsets) the "`disable`" option in the *removal* section.	overrides
`/link:[[+]<name>[,[+]<name>],]out=<name>`	Sets sub-elements of the `<linkedassembly>` element.	overrides
`/link:off`	Sets the "`disable`" option in the *linking* section.	overrides
`/premark: [on|off*|only]`	Sets (or unsets) the "`disable`" option in the *premark* section. The "`only`" setting is not saved to the configuration file.	overrides
`/watermark`	Sets the `<watermark>` element.	overrides
`/passphrase`	Sets the `<passphrase>` element and sets the usepassphrase option.	overrides
`/charmap`	Sets the "`encoding`" attribute in the *premark* section.	overrides

Examples:

The following examples use this configuration file that enables renaming with an output mapping file. It is referenced as "**myconfig.xml**" in the examples.

Example 1

```
<?xml version="1.0"?>
<!DOCTYPE dotfuscator SYSTEM
"http://www.preemptive.com/dotfuscator/dtd/dotfuscator_v2.2.dtd">
<dotfuscator version="2.2">
   <renaming>
      <mapping>
         <mapoutput overwrite="true">
            <file dir="${configdir}\reports" name="MyMap.xml"/>
         </mapoutput>
      </mapping>
   </renaming>
</dotfuscator>dotfuscator -in:my.dll myconfig.xml
```

This command specifies **my.dll** as an input assembly in library mode (because of the DLL extension), and applies the renaming options in the configuration file. In this case, control flow, string encryption, and pruning are disabled because they are implicitly disabled in the configuration file.

The output DLL will go in a directory called **".\Dotfuscated"**, since an output is not specified in the configuration file or on the command line.

Example 2

```
dotfuscator -in:my.dll -keep:namespace -enha:on -cont:high myconfig.xml
```

This command also specifies **my.dll** as an input assembly. In addition, it tells the renamer to keep namespaces and use enhanced overload induction. It also enables control flow obfuscation, setting the level to "high" for maximum obfuscation.

2.6.1.3 Saving a Configuration File from the Command Line

Once you have the command line settings that you want for your application, you can save a configuration file containing those settings by using the /makeconfig option. This takes all your command line options, merges them with your configuration template if you have one, and saves a custom configuration file that you can use for future runs.

Example

```
dotfuscator -in:my.dll -keep:namespace -enha:on -cont:high -make:new.xml myconfig.xml
```

The resulting configuration file (**`new.xml`**) shows the command line options merged with the options from the original configuration (**`myconfig.xml`**):

Configuration File

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!DOCTYPE dotfuscator SYSTEM
"http://www.preemptive.com/dotfuscator/dtd/dotfuscator_v2.2.dtd">
<dotfuscator version="2.2">
  <input>
    <asmlist>
      <inputassembly>
        <option>library</option>
        <file dir="." name="my.dll" />
      </inputassembly>
    </asmlist>
  </input>
  <output>
    <file dir="C:\MSProjects\dotfuscatortest\Dotfuscated" />
  </output>
  <renaming>
    <option>enhancedOI</option>
    <option>keepnamespace</option>
    <mapping>
      <mapoutput overwrite="true">
        <file dir="${configdir}\reports" name="MyMap.xml" />
      </mapoutput>
    </mapping>
  </renaming>
  <controlflow level="high" />
</dotfuscator>
```

2.6.1.4 Launching the Graphical User Interface from the Command Line

If you invoke Dotfuscator with no command line options, the standalone graphical user interface starts up with an empty project.

Alternatively, you can specify any combination of legal command line options along with the **`/g`** option to launch the user interface with those options in effect.

Example:

```
dotfuscator -g -in:my.dll myconfig.xml
```

This command launches the standalone user interface. The settings from **`myconfig.xml`** will be loaded, and **`my.dll`** will be set as an input assembly.

2.6.2 MSBuild Task Reference

Since version 2.0, the .NET Framework has shipped with a task driven build engine called MSBuild. Dotfuscator provides an MSBuild interface using tasks defined in PreEmptive.Dotfuscator.Tasks.dll. This DLL is installed into the following subdirectories of the MSBuild extensions directory…:

MSBuild Extensions Directory:

```
$(MSBuildExtensionsPath)\PreEmptive\Dotfuscator\<major>
$(MSBuildExtensionsPath)\PreEmptive\Dotfuscator\<major>\<major>.<minor>
$(MSBuildExtensionsPath)\PreEmptive\Dotfuscator\<major>\<major>.<minor>\<major>.<minor>.<patch>

For example if version 4.11.5 of Dotfuscator is installed you will see the following
directories:

$(MSBuildExtensionsPath)\PreEmptive\Dotfuscator\4
$(MSBuildExtensionsPath)\PreEmptive\Dotfuscator\4\4.11
$(MSBuildExtensionsPath)\PreEmptive\Dotfuscator\4\4.11\4.11.5
```

There is also a *targets* template that you can import into your MSBuild scripts, PreEmptive.Dotfuscator.Targets. This file is installed into the same directories.

The layout is designed so that when writing your own msbuild files you can point to the most specific version desired. For more information see the section on Side by Side Installs.

2.6.2.1 Dotfuscate Task

You can run Dotfuscator from MSBuild using the Dotfuscate task. Below are the properties provided by the Dotfuscate task.

Property	**Type**	**Description**
ConfigPath	**String**	**Write Only. Sets the path to the Dotfuscator configuration file.**
DebugSymbols	**String[]**	Read Only. Exposes PDB files associated with output assemblies. The **PreEmptive.Dotfuscator.Targets** file exposes this property as an output Item named **DotfuscatedDebugSymbols**.
InputAssemblies	**ITaskItem[]**	Write Only. Currently, the input assemblies must also

		be listed in the configuration file. This is only for interoperation with Visual Studio generated project files. This may change in the future.
MappingFile	**String**	Read Only. Exposes renaming map file. The PreEmptive.Dotfuscator.Targets file exposes this property as an output Item named **DotfuscatorMappingFile**.
OutputAssemblies	**String[]**	Read Only. Exposes output assemblies. The PreEmptive.Dotfuscator.Targets file exposes this property as an output Item named **DotfuscatedAssemblies**.
ReportFiles	**String[]**	Read Only. Exposes report files such as renaming HTML report and removal reports. The PreEmptive.Dotfuscator.Targets file exposes this property as an output Item named **DotfuscatorReportFiles**.
SatelliteAssemblies	**String[]**	Read Only. Exposes satellite assemblies associated with output assemblies. The PreEmptive.Dotfuscator.Targets file exposes this property as an output Item named **DotfuscatedSatelliteAssemblies**.

Properties	**String**	Write Only. Sets user defined external properties. The string must contain a valid XML element with child elements that represent key/value pairs.
		
		For example:
		
		`<Properties>`
		
		`<Property1>Value1</Property1>`
		
		`<Property2>2</Property2>`
		
		`</Properties>`

Dotfuscator Project Files in Visual Studio 2005 and later

If you are using the Visual Studio integrated version with Visual Studio 2005 and higher, your Dotfuscator project files are automatically persisted in MSBuild format, so you should be able to take your project tree over to a build machine using MSBuild without Visual Studio and have the build perform the same as it would inside Visual Studio.

2.6.2.2 PreMark Task

You can extract watermarks from previously watermarked assemblies using the PreMark task. This provides similar functionality to the command line program Premark.exe included in your Dotfuscator installation directory.

Property	Type	Description
InputAssemblyPath	**String**	Required, Read,Write. Gets or sets the path to the Assembly whose watermark you want to extract.
UsePassphrase	**Boolean**	Read,Write.
Passphrase	**String**	Read,Write. Currently, the input assemblies must also be listed in the configuration file. This is only for interoperation with Visual Studio generated project files. This may change in the future.
Watermark	**String**	Read Only. Exposes the extracted watermark string.

2.6.3 Configuration File Reference

Dotfuscator configuration files may have any name or extension, but usually have a `.xml` extension. Configuration files contain information about how a given application is to be Dotfuscated. The configuration file is an XML document conforming to `dotfuscator_v2.3.dtd` (or one of its predecessors), referenced in the appendix.

This section documents Dotfuscator’s XML configuration file. It contains detailed descriptions of each configuration option, making it useful as a reference, even if you are using Visual Studio, the standalone GUI, or the command line interface to generate a configuration file for you.

2.6.3.1 Version

The .xml file version attribute must be present and must be applicable to your version of Dotfuscator. It should match the version number of the DTD to which it conforms. Point releases of Dotfuscator are designed to be able to use unmodified configuration files from earlier versions. For example, you should be able to run Dotfuscator 1.1 using a version 1.0 configuration file without having to edit the configuration file.

Note: The configuration file version is not always the same as the version of Dotfuscator. Every version of Dotfuscator expects to see a specific, but not necessarily identical, version of the configuration file.

Version

```
<dotfuscator version="2.2">
```

2.6.3.2 Property List and Properties

The optional **`<propertylist>`** section allows for the definition and assignment of variables known as **`<properties>`** that may be used later in the configuration file. Property definitions defined in this section are referred to as *internal* properties.

Internal Properties

```
<!-- define expandable properties -->
<!-- optional -->
<propertylist>
  <property name="projectname" value="myproject"/>
  <property name="projectdir" value="c:\myprojects"/>
</propertylist>
```

Variables, or property references, may be used in the configuration file without being defined in this section. For example, they may be defined on the command line or come from the environment. Properties work via string substitution, using the following algorithm to find a value associated with the property:

1. Check the external property list for a value.
2. If not found, check for an environment variable with the same name as the property,
3. If not found, check for an internal definition in the propertylist section of the configuration file,
4. If still not found, use the empty string as the value.

External properties are passed in on the command line using the **-p** option. There are three built-in external properties:

- **applicationdir**, which reflects Dotfuscator’s installation directory.
- **appdatadir**, which reflects Dotfuscator’s local data directory.
- **configdir**, which reflects the directory in which the configuration file resides.

Properties are useful for creating configuration files that act as templates for multiple projects, for different versions of the same project, or for simple portability across different build environments. A property is referenced with the following syntax:

Property Syntax

```
${property_name}
```

Property references are case sensitive, therefore **${MyProjectDir}** references a different property than does **${myprojectdir}**. Currently, property references may only be used as values in the **dir** or **name** attributes of the **<file>** element, and not just anywhere in the configuration file. Here is a list of sections that use the **<file>** element:

inputassembly	mapinput	mapoutput
output	tempdir	assembly
removalreport	transform	key
loadpaths	program	filelist
smartobfuscationreport	pfx	

A property reference is interpreted literally in any other place in the configuration file. Property references may not be nested as doing so results in an error. Here is an example of a property reference in use:

Here is an example of a property reference in use:

```
<output>
  <file dir="${testdir}\output"/>
</output>
```

2.6.3.3 Global Section

The optional global section is for defining configuration options that apply across the entire run. This section describes each global option in detail.

Note: Global options are not case sensitive.

2.6.3.3.1 Library Global Option

This option was deprecated in Dotfuscator 3.0. It has been replaced with a more granular library option that can be applied to individual input assemblies. When reading older configuration files, Dotfuscator reads this option and honors it, but the user interface saves the configuration file using the new options. See Library Mode By Assembly.

Library Global Option

```
<global>
 <!—set library option -->
 <option>library</option>
</global>
```

2.6.3.3.2 Verbose, Quiet, and Investigate Global Options

These options are the same as the corresponding command line options. Setting the command line option sets the global option at run time. Alternatively, if the global option is set and the command line option is not set, then the global option takes precedence. In other words, there is no way to unset a set global option from the command line.

Verbose, Quiet, and Investigate Global Options

```
<global>
 <!-- run in verbose mode -->
 <option>verbose</option>
 <!-- run in quiet mode -->
 <option>quiet</option>
 <!-- investigate only and generate a map -->
 <option>investigate</option>
</global>
```

2.6.3.3.3 SuppressIldasmAttribute Global Option

Setting this option tells Dotfuscator to prevent Microsoft's Ildasm utility from displaying the assembly IL. This is only valid for assemblies targeting .NET 2.0 and above.

SuppressIldasm Option

```
<global>
   <option>suppressildasm</option>
</global>
```

2.6.3.3.4 Debug Global Option

Setting these options tells Dotfuscator to create a symbol file in PDB format for each output assembly. Debuggers use these files to provide useful information in a debugging session. Typically, they contain information such as line numbers, source file names, and local variable names. The PDB files are placed in the output directory with the output assemblies.

This option is useful for assemblies that already have associated PDB files (*i.e.* you told your compiler to generate debugging symbols). In this case, Dotfuscator uses information in the original PDB file to create the new PDB file for the obfuscated version. In this case, the line numbers and source file names correspond to the information in the original assembly.

If an input PDB file is missing, then the PDB files created by Dotfuscator will contain line numbers that correspond to the low level Microsoft Intermediate Language instructions in the obfuscated assembly.

Debug, DebugImpl, DebugOpt, and Pdb Global Options

```
<global>
   <!-- Disable JIT optimization,create PDB file, use sequence points from PDB -->
   <option>debug</option>
   <!-- Disable JIT optimization,create PDB file, use implicit sequence points -->
   <option>debugimpl</option>

   <!-- Enable JIT optimization, create PDB file, use implicit sequence points -->
   <option>debugopt</option>

   <!-- Create the PDB file without enabling debug info tracking -->
   <option>pdb</option>
</global>
```

DebugImpl, DebugOpt and Pdb global options are supported for assemblies targetting .Net 2.0 and above. If these options are used with assemblies targeting .Net 1.0 or 1.1, Debug global option will be used instead.

2.6.3.3.5 NoDotfuscatorAttribute Global Option

Dotfuscator, by default, inserts a custom attribute into your application named **`DotfuscatorAttribute`**. The attribute contains information about the Dotfuscator version used to obfuscate the program, including product ID (CE vs. PE) and version numbers. The purpose is threefold:

- To identify which Dotfuscator version was used on a program.
- Future Dotfuscator versions will use this information to identify already obfuscated third-party assemblies used as inputs, as special treatment may be required in these cases.
- PreEmptive Solutions is actively working with other tool vendors to ease the pain of debugging obfuscated code; the attribute helps those tools know what they are dealing with.

If these things are not as important to you as the increased security of an anonymously obfuscated program, or if size reduction is a higher priority, then you can disable insertion of the attribute by manually setting a global option in the configuration file, called **`nodotfuscatorattribute`**.

NoDotfuscatorAttribute Global Option

```
<global>
   <option>nodotfuscatorattribute</option>
</global>
```

2.6.3.4 Input Assembly List

The input assembly list contains the file names and directories of the assemblies and/or packages you want to Dotfuscate. It also contains configuration options that are set at the package or assembly level.

If you have a multi-module assembly, only list the module containing the manifest.

Example Title

```
<input>
   <asmlist>
      <inputassembly>
       ...
        <file dir="c:\temp" name="myproj.dll"/>
       </inputassembly>
     ...
   </asmlist>
</input>
```

2.6.3.4.1 Library Mode By Assembly

This setting tells Dotfuscator that a particular input assembly constitutes a library. For Dotfuscation purposes, a library is defined as an assembly that is referenced from other components not specified as one of the inputs in this run. This has implications for renaming and pruning, regardless of any custom excludes you may have set.

Here are the rules when using the library option:

- Names of public classes and nested public classes are not renamed. Members (fields and methods) of these classes are also not renamed if they have public, family, or famorassem access.
- In addition, no virtual methods are renamed, regardless of access specifier. This allows clients of your library to override private virtual methods if necessary.
- Any user-specified custom renaming exclusions are applied in addition to the exclusions implied by the above rules.
- Property and Event metadata are always preserved.
- Pruning Rules.
  - Public classes are not removed, even if static analysis determines they are not required.
  - Fields of these classes are not removed if they have public, family, or famorassem access.
  - Methods of these classes are not removed if they have public, family, or famorassem access. In addition, such methods are treated as entry points, so their call trees are followed and all subsequently called methods are also protected from removal.

If you do not have the library option set for an assembly, then you are telling Dotfuscator that your input assembly is a standalone application, or that it will only be referenced by other input assemblies. In this case obfuscation is more aggressive:

- Everything is renamed except methods that override classes external to the application (*i.e.* classes in assemblies that are not included in the run.)
- Property and Event metadata is removed, since this metadata is not required to run the application.
- User specified custom renaming exclusions are also applied.

- Pruning Rules
  - Specifically included classes, methods, or fields are not pruned.
  - All trigger methods and fields are not pruned.
  - All classes, members, and fields that are excluded from renaming also become excluded from pruning.
  - All other classes, fields, and methods that are unreachable from some included class, method, or field are pruned.

To specify library mode for an input assembly, add an **`<option>`** element to its **`<inputassembly>`** element.

Library Mode by Assembly

```
      <inputassembly>
        <option>library</option>
        <file dir="c:\temp" name="myproj.dll"/>
      </inputassembly>
```

2.6.3.4.2 Declarative Obfuscation By Assembly

For a complete description of Dotfuscator's support for Declarative Obfuscation, see Declarative Obfuscation via Custom Attributes.

Enabling or Disabling Declarative Obfuscation

Dotfuscator allows you to switch Declarative Obfuscation on or off for specific input assemblies. If not enabled, Dotfuscator ignores obfuscation related custom attributes.

To enable declarative obfuscation via the configuration file, you add a **`honorOAs`** option to each **`<inputassembly>`** element.

honorOAs Option Added to an Input Assembly Element

```
      <inputassembly>
        <option>honoroas</option>
        ...
      </inputassembly>
```

Stripping Declarative Obfuscation Attributes

To tell Dotfuscator to strip obfuscation attributes via the configuration file, you add a **`honorOAs`** option to each **`<inputassembly>`** element.

stripOAs Option Added to an Input Assembly Element

```
      <inputassembly>
        <option>stripoa</option>
        ...
      </inputassembly>
```

2.6.3.4.3 Instrumentation Processing By Assembly

Dotfuscator allows you to control instrumentation for specific input assemblies. The section on Configuring and Running Dotfuscator with Application Analytics describes the effects of setting these options.

When any of the instrumenting transforms are enabled, Dotfuscator honors and strips unnecessary instrumentation attributes from the output assemblies. The default behavior can be overridden at the input assembly level by specifying one or both:

- **nohonorsos**, to prevent Dotfuscator from acting on instrumentation attribute directives.
- **nostripsos**, to prevent Dotfuscator from stripping unnecessary instrumentation attributes.

Instrumentation Processing by Assembly

```
      <inputassembly>
        <!-- do not strip instrumentation attributes -->
        <option>nostripsos</option>
        <!-- do not honor instrumentation attributes -->
        <option>nohonorsos</option>
        ...
      </inputassembly>
```

2.6.3.4.4 Transform XAML By Assembly

This setting tells Dotfuscator that a particular input assembly may contain markup, either XAML as found in Silverlight applications or compiled XAML resources (BAML) as found in Windows Presentation Foundation applications; and any markup should be analyzed and included for renaming. For Dotfuscation purposes, markup that is transformed will have identifiers renamed in conjunction with any code-behind references of the elements. Properties that are referenced from markup resources will have their property metadata retained but will be renamed.

To specify Transform XAML mode for an input assembly, add an **<option>** element to its **<inputassembly>** element.

Transform XAML Mode by Assembly

```
      <inputassembly>
        <option>transformxaml</option>
        <file dir="c:\temp" name="myproj.dll"/>
      </inputassembly>
```

2.6.3.5 User Defined Assembly Load Path

Dotfuscator needs to load assemblies referenced by your input assemblies in order to discover information about types you are using in your input assemblies. Dotfuscator uses discovery rules similar to the rules used by Visual Studio and the CLR itself.

If a referenced assembly cannot be found using the default search rules, Dotfuscator provides a way for you to specify additional directories in which to look for referenced assemblies. Dotfuscator searches these directories in the specified order as the last step in its algorithm. However, if the prepend option (the **Search First** checkbox in the Project Properties and Settings Tab) is used, then Dotfuscator searches the load path before applying its standard search.

To add a **User Defined Assembly Load Path** to your XML configuration file:

Adding a User Defined Assembly Load Path

```
  <input>
    <loadpaths>
      <option>prepend</option>
      <file dir="C:\temp" />
       ...
    </loadpaths>
    ....
  </input>
```

2.6.3.6 Output Directory

This is the directory where output assemblies are written. The application always overwrites files in this directory without prompting the user.

Output Directory

```
<!-- destination directory is required -->
<output>
<file dir="c:\work"/>
</output>
```

2.6.3.7 Temp Directory

This section is optional and specifies Dotfuscator’s working directory. If not specified, the working directory defaults to the system’s temporary directory. The application uses the working directory to run ildasm and ilasm on the input assemblies. The disassembled output is stored in this directory along with any resources embedded in the input assemblies. These files are automatically deleted after processing.

Temp Directory

```
<!-- scratch directory is optional -->
<!-- If absent, defaults to system's temp dir -->
<tempdir>
<file dir="c:\temp"/>
</tempdir>
```

2.6.3.8 Obfuscation Attribute Feature Map

The feature map is for Declarative Obfuscation. For a complete description of Dotfuscator's support for Declarative Obfuscation, see Declarative Obfuscation via Custom Attributes. That section describes the Feature Map and lists the native feature strings that Dotfuscator understands.

In the configuration file, the **`<obfuscationattributemap>`** element is where you can map strings obtained from an obfuscation attribute's Feature property to one or more feature strings that Dotfuscator understands.

Here is what such a mapping looks like in the XML configuration file:

Obfuscation Attribute Feature Map

```
<obfuscationattributemap>
  <feature name="testmode">renaming, controlflow</feature>
</obfuscationattributemap>
```

2.6.3.9 Renaming Section

The renaming section allows you to specify options that are specific to renaming, input and output mapping file locations, and fine-grained rules for excluding items from renaming.

The renaming section is optional. If not present, the following defaults apply:

- Default renaming (namespaces are removed).
- New names are chosen using the **`loweralpha`** renaming scheme.
- No mapping file read or written.
- No exclusions beyond those dictated by your application type.

The section on identifier renaming describes the renaming options, the mapping file, and custom exclusions in great depth. The following sections present an overview of each.

2.6.3.9.1 Renaming Scheme

Dotfuscator allows you to choose from several predefined algorithms for generating obfuscated identifier names:

- **Lower Alpha**. This is the default renaming scheme used by all versions of Dotfuscator: **`{a,b,c,...}`**
- **Upper Alpha**. This scheme uses upper case alphanumeric characters: **`{A,B,C,...}`**
- **Numeric**. This scheme uses numeric characters only: **`{0,1,2,...}`**
- **Unprintable**. This scheme uses unprintable high Unicode code points.

The renaming scheme is an attribute of the **`<renaming>`** element. Allowable values are: **`loweralpha`**, **`upperalpha`**, **`numeric`**, or **`unprintable`**.

Renaming Scheme

```
<renaming scheme="unprintable">
...
</renaming>
```

2.6.3.9.2 Renaming Options

Dotfuscator allows several options that govern how **namespaces** are treated by the renaming algorithm. These are: "**keepnamespace**" and "**keephierarchy**" and are explained in detail in the section on identifier renaming.

The **disable** option is primarily for convenience and troubleshooting purposes. When set, Dotfuscator skips renaming altogether, regardless of what's in the rest of the renaming section.

Renaming Options - Keepnamespace

```
<renaming>
<!--  Keep namespaces as they are, but rename types. -->
<option>keepnamespace</option>

<!-- Preserves namespace hierarchy but rename -->
<!-- namespace names -->
<option>keephierarchy</option>

<!-- Skip renaming, ignoring rest of section -->
<option>disable</option>
...
</renaming>
```

Dotfuscator allows you to specify that obfuscated type names must be prefixed with a default or user-specified string. To use this feature, specify the **prefix** option. Please see Renaming Prefixes for full details.

Turning on the Renaming Prefix Feature

```
<renaming>
   <!-- this turns on the renaming prefix feature -->
   <option>prefix</option>
...
</renaming>
```

Dotfuscator also allows an enhanced level of Overload Induction that adds return type to the mix. The option to turn this on is: **enhancedOI** and is explained in detail in the section on overload induction method renaming.

Apply Enhanced Overload Induction

```
<renaming> <!--  Apply Enhanced Overload Induction. -->
 <option>enhancedOI</option>
 ...
</renaming>
```

Enhanced Overload Induction is, by default, not applied to classes marked as serializable. If you wish to apply enhanced overload induction to all types, including serializable types, use the **`enhancedOIOnSerializables`** option:

Apply Enhanced Overload Induction Serializeable Option

```
<renaming>

<!--  Apply enhanced Overload Induction even on serializable types. -->
<option>enhancedOIOnSerializables</option>
...
</renaming>
```

You can change the renaming algorithm to rename types and members in a way that's compatible with the XML Serializer.

Change Renaming Algorithm to be Compatible with XML Serializer

```
<renaming>

<!--  XML Serialization compatibility. -->
<option>xmlserialization</option>
...
</renaming>
```

For more information, see XML Serialization and Renaming.

Using the **`<explicitoverrides>`** renaming option lets Dotfuscator rename more methods by allowing it to introduce explicit (i.e. non-syntactic) method overrides. In other words, overridden methods can have different names than the methods they override. For example, ordinarily, if a method overrides Object.ToString(), Dotfuscator would not be able to rename it without breaking the override relationship, since typically the Object class is not in an input assembly and therefore its ToString() would not be renamed. With this setting, Dotfuscator can rename the overriding method and introduce metadata that tells the CLR that the method is meant to override Object.ToString(). For more information, see Introduce Explicit Method Overrides When Renaming.

Introduce Explicit Method Overrides

```
<renaming>
<!-- Allow overriding methods to have different names than the
       methods they override -->
   <option>explicitoverrides</option>
</renaming>
```

If you want to rename overloaded methods with the same name, then you would not use the **`<explicitoverrides>`** renaming option.

Note: Renaming options are not case sensitive.

2.6.3.9.3 Renaming Exclusion List

This section provides a dynamic way to fine tune the renaming of the input assemblies. It can contain a list of exclusion rules that are applied at runtime. If a rule selects a given class, method, field, property, or event then that item is not renamed.

- These rules are applied *in addition to* rules implied by global options such as the library option.
- The rules are logically **`OR`**-ed together, so any item that is selected by at least one rule is not renamed.
- The exclusion list has support for excluding names by type, method, field, property, event, assembly, module, or namespace.
- Each type of rule is explained in detail in the section on identifier renaming.

2.6.3.9.4 Renaming Referenced Rules

Referenced rules allow you to import rules from an external file so they can be shared among configurations. Dotfuscator's Built-In renaming rules use this to import rules from **%ProgramData%\PreEmptive Solutions\Common\dotfuscatorReferenceRule_v1.1.xml**. The rule is referenced via the **`rulekey`** attribute whose value is a GUID defined by the rule being referenced.

Reference Rule List

```
<referencerulelist>
   <referencerule rulekey="{0D471A86-E98F-4493-849B-85BD4CC884A1}"/>
   <referencerule rulekey="{C9D9BF84-4F0D-4e9f-B3EC-3038235AE741}"/>
</referencerulelist>
```

2.6.3.9.5 Output Mapping File

This feature of Dotfuscator produces a log of all the renaming mappings used by Dotfuscator during a specific run. It also provides a statistics section.

Specifying this option instructs Dotfuscator's renamer to keep track of how things were renamed for both your immediate review and for possible use as input in a future Dotfuscator run. A file is created from this option that is then used in the incremental input file option.

Accidental loss of this file can destroy your chances of incrementally updating your application in the future. Therefore, proper backup of this file is crucial. For this reason, Dotfuscator automatically renames an existing map file with the same name before overwriting it with a new version of the map file.

If you do not want Dotfuscator to rename existing map files before overwriting, set the attribute **`overwrite="true"`**.

The format of the mapping file is discussed in the section on identifier renaming.

Setting the Attribute to Overwrite

```
<renaming>
...
<mapping>
  <mapoutput overwrite="true">
   <file dir="c:\work" name="testout.xml"/>
  </mapoutput>
</mapping>
</renaming>
```

2.6.3.9.6 HTML Renaming Report

The output mapping file is natively formatted as an XML document suitable for parsing. For a human readable renaming report, you can tell Dotfuscator to transform the output mapping file into an HTML formatted document. Dotfuscator will apply a predefined XSL document to the mapping file to accomplish the transformation. If you do not like the default HTML report, you can optionally specify your own XSL document to use for the transformation. The output report is placed in the same directory as the XML formatted mapping file. The filename will be the same as the XML file, but will have a **`.html`** extension rather than a **`.xml`** extension.

Example Title

```
<renaming>
...
<mapping>
  <mapoutput overwrite="true">
   <file dir="c:\work" name="testout.xml"/>
   <transform>
      <!-- specifying your own XSL file is optional -->
      <file dir="c:\mytransforms" name="map.xsl"/>
   </transform>
  </mapoutput>
</mapping>
</renaming>
```

2.6.3.9.7 Input Mapping File

In Dotfuscator, the input mapping file allows you to import names that Dotfuscator created in a previous run (a process known as Incremental Obfuscation). Dotfuscator will make a best-effort attempt to rename classes, methods, and fields to the names indicated in the input mapping file.

The **<mapinput>** element allows you to specify the input mapping file. It also has an optional **obfuscatereferences** attribute, which defaults to "**true**" if not present. This attribute controls how Dotfuscator handles names contained in the input mapping file that are not defined within the set of input assemblies. When true, references to these names within the current set of input assemblies will be renamed.

Input Mapping File

```
<renaming>
...
<mapping>
  <mapinput obfuscatereferences="true">
   <file dir="c:\work" name="testin.xml"/>
  </mapinput>
</mapping>
</renaming>
```

2.6.3.10 Control Flow Obfuscation Section

The control flow section allows you to specify options that are specific to control flow obfuscation, including fine-grained rules for excluding items from control flow obfuscation.

The control flow section is optional. If not present, control flow obfuscation is disabled.

2.6.3.10.1 Control Flow Obfuscation Level

The level of control flow obfuscation may be set to one of three values: "**low**", "**medium**", or "**high**". These levels correspond to the aggressiveness of Dotfuscator’s control flow obfuscation algorithms. A higher level generally results in stronger obfuscation at the cost of increased code size and degraded performance. This is because more aggressive control flow obfuscation involves adding more branch instructions to the code.

The level of control flow obfuscation applies globally to all methods being obfuscated.

2.6.3.10.2 Control Flow Obfuscation Options

The "**disable**" option is primarily for convenience and troubleshooting purposes. When set, Dotfuscator skips control flow obfuscation altogether, regardless of what’s in the rest of the control flow section.

Example Title

```
<controlflow level="high">
<!-- Skip control flow, ignoring rest of section-->
<option>disable</option>
...
</controlflow>
```

Note: Control Flow options are not case sensitive.

2.6.3.10.3 Control Flow Exclusion List

This section provides a dynamic way to fine tune control flow obfuscation of the input assemblies. It can contain a list of exclusion rules that are applied at runtime. If a rule selects a given class or method, then that item is not subject to control flow obfuscation.

Note: The library option has no effect on control flow obfuscation, unlike renaming.

The rules are logically **`OR`**-ed together, so any item selected by at least one rule is not subject to control flow obfuscation.

The exclusion list has support for excluding methods by type, method, assembly, module, or namespace.

Each type of rule is explained in detail in the section on control flow obfuscation.

2.6.3.11 String Encryption Section

The string encryption section allows you to specify options that are specific to string encryption, including fine-grained rules for specifying types and methods subject to string encryption.

The string encryption section is optional. If not present, string encryption is disabled.

2.6.3.11.1 String Encryption Options

This option is primarily used for convenience and troubleshooting purposes. When set, Dotfuscator skips string encryption altogether, regardless of what's in the rest of the string encryption section.

User String Encryption Options

```
<stringencrypt>
<!--Skip string encryption, ignoring rest of section-->
<option>disable</option>
...
</stringencrypt>
```

Note: String Encryption options are not case sensitive.

2.6.3.11.2 String Encryption Inclusion List

This section provides a dynamic way to fine tune string encryption of the input assemblies. It contains a list of inclusion rules that are applied at runtime. If a rule selects a given class or method, then that item is subject to string encryption.

> **Note**: Unlike renaming, the library option has no effect on string encryption.

The rules are logically `OR`-ed together, so any item that is selected by at least one rule is subject to string encryption.

The inclusion list has support for including methods by type, method, assembly, module, or namespace.

Each type of rule is explained in detail in the section on user string encryption.

2.6.3.12 Removal Section (i.e. Pruning)

The **`<removal>`** section allows you to specify options that are specific to the pruning functionality and fine-grained rules for specifying types and members subject to pruning.

The **`<removal>`** section is optional. If not present, removal is disabled and no pruning occurs.

2.6.3.12.1 Disable Removal Option

This option is used primarily for convenience and troubleshooting purposes. When set, Dotfuscator skips the pruning step altogether, regardless of what's in the rest of removal section.

Disable Removal Option

```
<removal>
<!--Skip removal (pruning), ignoring rest of section-->
<option>disable</option>
...
</removal>
```

> **Note**: Removal options are not case sensitive.

2.6.3.12.2 ConstOnly Option

This option is used to enable Constant-Only pruning. In this mode, only constant declarations will be pruned. Unused types, methods, and fields will be propagated to the output assembly.

Disable Removal Option

```
<removal>
<!--Use Constant-Only pruning instead of full pruning-->
<option>constonly</option>
...
</removal>
```

Note: Removal options are not case sensitive.

2.6.3.12.3 Removal Trigger List

In the context of pruning, *triggers* are starting points for the static dependency analysis that Dotfuscator performs in order to determine which types, methods, and fields are used by your code. In other words, these are the *entry points* for your application or library.

Triggers are analyzed by Dotfuscator to determine which classes, methods, and fields are required for your application or library to function. For example, all methods called by your triggers, and methods called by those methods, are deemed required by Dotfuscator. That is, if you tell Dotfuscator a specific main method is required, then all the methods that main method calls are required as well.

A trigger list is not required, but is honored, if the library global option is used.

The trigger list is used *in addition to* triggers implied by global options such as the library option.

The trigger list specifies triggers in the same way that elements are selected for exclusion and inclusion in other parts of the configuration. It contains a list of rules that are applied at runtime. If a rule selects a given method or field, then that becomes a trigger.

The rules are logically **`OR`**-ed together, so any item that is selected by at least one rule becomes a trigger.

The trigger list has support for specifying fields, methods, properties, and events by type, method, assembly, module, or namespace.

Each type of rule is explained in detail in the section on pruning.

2.6.3.12.4 Conditional Includes List

A type must be conditionally included if it is not detectable by the static dependency analysis, *i.e.* if it is dynamically loaded; meaning, the type itself is included in the dependency analysis, but its members are still subject to pruning. Please see Understanding Include Triggers and Conditional Includes for a deeper explanation of this feature.

This section provides a dynamic way to specify conditionally included types. It contains a list of inclusion rules that are applied at runtime. If a rule selects a given type, then that item is conditionally included.

The rules are logically **`OR`**-ed together, so any item that is selected by at least one rule will be conditionally included.

The inclusion list has support for selecting types by name, assembly, module, or namespace.

Each type of rule is explained in detail in the section on pruning.

2.6.3.12.5 Removal Referenced Rules

Referenced rules allow you to import rules from an external file so they can be shared among configurations. Dotfuscator's Built-In removal rules use this to import rules from **%ProgramData%\PreEmptive Solutions\Common\dotfuscatorReferenceRule_v1.4.xml**. The rule is referenced via the **`rulekey`** attribute whose value is a GUID defined by the rule being referenced.

Referenced Rules

```
<referencerulelist>
   <referencerule rulekey="{0D458786-E99F-4593-849B-8512493884A1}"/>
   <referencerule rulekey="{C1159284-4F0D-4e9f-B3EC-303828419741}"/>
</referencerulelist>
```

2.6.3.12.6 Removal Report

The removal report provides a summary of all the elements removed by Dotfuscator during a specific run, including a statistics section. Dotfuscator writes the removal report in a parsable and easily transformed XML format. Like the renaming map file, Dotfuscator has a default transform that can generate a human readable HTML formatted version of the report.

Dotfuscator automatically renames an existing removal report with the same name before overwriting it with a new version.

If you do not want Dotfuscator to rename existing removal reports before overwriting, set the attribute **`overwrite="true".`**

The XML format of the removal report file is discussed in the section on pruning.

Removal Report

```
<removal>
<removalreport overwrite="true">
  <file dir="c:\work" name="report.xml"/>
  <transform>
     <!-- specifying your own XSL file is optional -->
     <file dir="c:\mytransforms" name="removal.xsl"/>
  </transform>
</removalreport>
</removal>
```

2.6.3.13 Linking Section

The linking section allows you to specify options that are specific to the assembly linking. For more information about the linker, see Assembly Linking.

The linking section is optional. If not present, assembly linking is disabled.

2.6.3.13.1 Disable Linking Option

This option is used primarily for convenience and troubleshooting purposes. When set, Dotfuscator skips the linking step altogether, regardless of what's in the rest of the linking section.

Disable Linking Option

```
<linking>
<!--Skip linking, ignoring rest of section-->
<option>disable</option>
...
</linking>
```

Note: Linking options are not case sensitive.

2.6.3.13.2 Linked Assemblies

A **`<linkedassembly>`** element specifies that one or more input assemblies should be linked into a specified output assembly. The linking section can contain multiple **`<linkedassembly>`** elements; therefore, you can use the linker to create multiple output assemblies. The only limitation is that you cannot link the same input assembly into multiple output assemblies.

The **`<linkedassembly>`** element contains sub-elements that allow you to specify options for the link step, such as the name mangling policy, the primary assembly, the list of input assemblies, and the name of the output assembly.

Options

Currently the only option you can specify is the name mangling policy, which may be one of:

- donotmangle
- manglesilently
- mangleandwarn

Primary Assembly

The **`<primaryinput>`** element identifies the prime assembly whose manifest information is used to create the output assembly's manifest. This assembly must also be listed in the subsequent **`<assemblylist>`** element.

Assemblies to Link

The assemblies you want to link are listed using an **`<assemblylist>`** element. Each must also be listed as an input assembly.

Output Assembly

The **`<outputassembly>`** element allows you to specify a name for the output assembly and an optional entry point method. The assembly is written to the destination directory with the given name.

Example

The example **`<linkedassembly>`** element specifies that input assemblies **`Driver.exe`** and **`LibraryC.dll`** should be linked into an assembly named **`out.exe`**. The entry point method is explicitly set to the **`Main`** method in the **`Driver`** assembly.

Note: An entry point only needs to be explicitly specified in ambiguous cases. For example, the output assembly is an .exe and the input assemblies contain more than one entry point.

Linked Assembly

```
    <linkedassembly>
      <option>donotmangle</option>
      <primaryinput>
        <assembly>
          <file dir="${configdir}" name="Driver.exe" />
        </assembly>
      </primaryinput>
      <assemblylist>
        <assembly>
          <file dir="${configdir}" name="Driver.exe" />
        </assembly>
        <assembly>
          <file dir="${configdir}" name="LibraryC.dll" />
        </assembly>
      </assemblylist>
      <outputassembly name="out.exe">
        <entrypoint>
          <type name="Driver.Form1">
            <method name="Main" signature="" />
          </type>
        </entrypoint>
      </outputassembly>
    </linkedassembly>
```

2.6.3.14 PreMark Section

The premark section allows you to specify options that are specific to watermarking. For more information about watermarking assemblies, see Watermarking.

Note: The premark section is optional. If not present, watermarking is disabled.

2.6.3.14.1 PreMark Options

The watermarker supports several configuration options.

The **`usepassphrase`** option tells the watermarker to encrypt the watermark string before applying it to the selected assemblies.

The **`truncatestring`** option tells the watermarker to truncate the watermark string if it will not fit in a given assembly. The default action is to halt the build if the watermark string is too large. See Watermark String Length for more details.

The **disable** option is used primarily for convenience and troubleshooting purposes. When set, Dotfuscator skips the watermarking step altogether, regardless of what's in the rest of the premark section.

PreMark Options

```
<premark>
<!--Skip watermarking, ignoring rest of section-->
<option>disable</option>
<!--Encrypt the watermark string -->
<option>usepassphrase</option>
<!--Truncate the string and continue if the string is too big -->
<option>truncatestring</option>
...
</premark>
```

Note: PreMark options are not case sensitive.

2.6.3.14.2 PreMark Elements

The premark section contains sub-elements that allow you to specify which input assemblies you would like to watermark, a passphrase to use if you are encrypting your watermark string, a character encoding to use, and the watermark string itself.

Assemblies to Watermark

The assemblies you want to watermark are listed using an **<assemblylist>** element. Each must also be listed as an input assembly.

<passphrase>

The **<passphrase>** element specifies the passphrase to use when encrypting the watermark string. The watermark string will be encrypted if you have set the **usepassphrase** option and the passphrase itself is set.

<encoding>

The **<encoding>** element specifies the character encoding, or character map, to use when encoding the watermark string. The **<encoding>** element has a **name** attribute that you can set to a supported character map. Character maps and their names are described in the Character Maps section.

<watermark>

The **<watermark>** element specifies the string that you want to apply to the selected assemblies. The string is first encoded using the character map, then optionally encrypted.

Example

This example configures PreMark to apply a watermark to **`MyApp.exe`**. The watermark string, **`MY WATERMARK`** is first encoded using the **`6bit-a`** character map, then encrypted with the passphrase "**`mommy`**".

PreMark Elements

```
<premark>
  <option>usepassphrase</option>
  <option>truncatestring</option>
  <assemblylist>
    <assembly>
      <file dir="${configdir}" name="MyApp.exe" />
    </assembly>
  </assemblylist>
  <passphrase>mommy</passphrase>
  <encoding name="6bit-a" />
  <watermark>MY WATERMARK</watermark>
</premark>
```

2.6.3.15 Signing Section

The signing section allows you to specify if and how you want Dotfuscator to sign your strongly named output assemblies. For more information about assembly signing, see Dotfuscating Strong Named Assemblies.

The signing section is optional. If not present, your strongly named input assemblies will not be resigned after Dotfuscation, and Authenticode Digital Signing will not be applied.

Specifying <key> Element

You can specify the key or key pair that you want Dotfuscator to use when signing with a **`<key>`** element. A **`<key>`** element can contain either a **`<file>`** or a **`<container>`** sub element. A **`<file>`** element references the file containing the key or key pair. A **`<container>`** element has a "**`name`**" attribute that specifies the name of the key container.

<file> Element

```
    <key>
      <file dir="c:\temp" name="key.snk" />
    </key>
```

<container> Element

```
    <key>
      <container name="foo"/>
    </key>
```

<resign> Element

To resign an assembly that was already signed before Dotfuscation, use a **`<resign>`** element. If the assembly has custom attributes that specify the key to use, you do not need a **`<key>`** element. If you wish to ignore the attributes, set the **`dontuseattributes`** option and provide a **`<key>`** element. If our assembly does not have custom attributes that specify the key, you must provide a **`<key>`** element.

This example tells Dotfuscator to ignore any custom attributes that specify the key and instead manually specifies a key file.

<resign> Element

```
<signing>
  <resign>
    <option>dontuseattributes</option>
    <key>
      <file dir="c:\temp" name="key.snk" />
    </key>
  </resign>
  ...
</signing>
```

<delaysign> Element

If your input assembly is delay signed and you want Dotfuscator to automatically complete the signing process, you can provide a **`<delaysign>`** element with a **`<key>`** sub-element.

<delaysign> Element

```
<signing>
  ...
  <delaysign>
    <key>
      <file dir="c:\temp" name="key.snk" />
    </key>
  </delaysign>
</signing>
```

2.6.3.16 Digital Signing Section

The digitalsigning section allows you to specify if and how you want Dotfuscator to perform Authenticode Digital Signing on your output assemblies.

The digitalsigning section is optional. If not present, Authenticode signing will not be applied.

Specifying <pfx> Element

You can specify the PKCS #12 file that was provided to you by your code-signing authority for use in Authenticode signing with a **`<pfx>`** element. A **`<pfx>`** element contains a **`<file>`** sub element which describes the location of the PKCS #12 (.pfx) file containing your code-signing certificate. The **`<pfx>`** element also has a **`password`** attribute which specifies the password used to unlock the certificate.

<pfx> Element

```
      <pfx password="secret123">
        <file dir="c:\temp" name="authenticode.pfx" />
      </pfx>
```

<digitalsigning> Element

To perform Authenticode signing on output assemblies, you can provide a **`<digitalsigning>`** element with a **`<pfx>`** sub-element.

The **`disable`** option is primarily for convenience and troubleshooting purposes. When set, Dotfuscator skips Authenticode signing altogether, regardless of what's in the rest of the signing section.

The **`<timestampurl>`** sub element provides the ability for you to specify the URL of an Authenticode timestamp service. This URL will be accessed during Dotfuscator's signing process, and will provide additional data which will allow your assemblies' Authenticode signatures to remain valid after your code-signing certificate has expired. This element is optional. If omitted, this additional data will not be included, and your assemblies' Authenticode signatures will become invalid once your code-signing certificate expires.

<digitalsigning> Element

```
  <digitalsigning>
    <!--Skip Authenticode signing, ignoring rest of section-->
    <option>disable</option>
    <!--Specify the certificate to use for Authenticode signing-->
    <pfx password="secret123">
      <file dir="c:\temp" name="authenticode.pfx" />
    </pfx>
    <!--Optionally specify the URL to the timestamp service-->
    <timestampurl>http://timestamp.comodoca.com/authenticode</timestampurl>
  </digitalsigning>
```

2.6.3.17 EventList Section

The eventlist section allows you to specify pre and post build events. For more information about Dotfuscator's build events, see Build Events.

The eventlist section is optional.

<event> Element

An event is essentially a program that Dotfuscator runs at a particular point in its build sequence. You can specify the program name, working directory, and command line arguments. In addition, an event can support additional configuration settings using an **`<option>`** element.

An **`<event>`** element has a **`type`** attribute. Currently Dotfuscator understands two event types: **`prebuild`** and **`postbuild`**. The program to run is specified using a **`<program>`** element.

Event <program> Element

The program to run when an event occurs is specified with the **`<program>`** element. It contains a **`<file>`** element specifying the program and its location, as well as an **`<environment>`** element that specifies command line arguments and working directory.

Like files and directories, the **`commandline`** attribute can contain property macros.

Property Macros

```
      <program>
        <file dir="c\temp" name="copyfiles.bat" />
        <environment commandline="${myproperty}" workingdir="c:\temp" />
      </program>
```

Pre-Build Event Options

The pre-build event supports one option that can be set using an **`<option>`** element nested within the **`<event>`** element for the pre build event.

Option	
haltonfail	If the build event program returns a non-zero error code, halt the Dotfuscator build.

Post-Build Event Options

The post-build event supports several options that can be set using an **`<option>`** element nested within the **`<event>`** element for the post build event.

Option	
haltonfail	If the build event program returns a non-zero error code, halt the Dotfuscator build.
runoneachmodule	Run the post build event once for each output module.
always	Run the post build event all the time, regardless of the success or failure of the

	Dotfuscator build.
buildfails	Run the post build event only when the Dotfuscator build fails.
buildsuccessful	Run the post build event only when the Dotfuscator build is successful.

Example

The following example shows an XML configuration file fragment that sets up pre- and post-build events. The pre-build event executes the program **c:\temp\copyfiles.bat** with no arguments. If the build is successful, then the post-build event executes the program **PEVerify** on each output assembly. Notice that the output assembly name is passed as a property to **PEVerify**.

Example XML Configuration File Fragment

```
  <eventlist>
    <event type="prebuild">
      <program>
        <file dir="c\temp" name="copyfiles.bat" />
        <environment commandline="" workingdir="c:\temp" />
      </program>
    </event>
    <event type="postbuild">
      <option>runoneachmodule</option>
      <option>haltonfail</option>
      <program>
        <file dir="C:\Program Files\Microsoft Visual Studio .NET 2003\SDK\v1.1\Bin"
name="PEVerify.exe" />
        <environment commandline="${dotf.current.out.module}"
                     workingdir="${dotf.destination}" />
      </program>
    </event>
  </eventlist>
```

2.6.3.18 PreEmptive Analytics Section

The PreEmptive Analytics section allows you to specify options that control how Dotfuscator processes instrumentation attributes in the input assemblies. For more information about instrumentation and PreEmptive Analytics, see Instrumentation (Tamper, Shelf Life, Exception, Analytics).

The PreEmptive Analytics section is optional.

The **sos** element defines a **mergeruntime** attribute, which can be **true** (default) or **false**. When **true**, Dotfuscator merges the PreEmptive Analytics library into one of the input assemblies. When **false**, Dotfuscator writes the runtime library DLL to the destination directory along with the other input assemblies.

There are four additional options for PreEmptive Analytics: **Enable PreEmptive Analytics**, **Send Analytics Messages**, **Don’t Send Tamper Messages**, and **Send Shelf Life Messages**.

The **sendanalytics** option tells Dotfuscator to add code to marked assemblies that sends application analytics startup, shutdown, and feature messages to a PreEmptive Analytics Endpoint. Unsetting this option is useful if you want to send tamper messages without analytics.

The **dontsendtamper** option tells Dotfuscator not to send tamper notifications if tampering is detected. This is useful if you want to detect tampering and have your application react locally, but you do not want the application to send messages to a PreEmptive Analytics Endpoint.

The **sendshelflife** option tells Dotfuscator to add code to marked assemblies that adds logic to send shelf life status messages. These include the warning, expiration, and sign of life messages.

The **disable** option is used primarily for convenience and troubleshooting purposes. When set, Dotfuscator skips the attribute processing altogether, regardless of what attributes are in the input assemblies and what options are in the rest of the sos section.

PreEmptive Analytics Processing

```
  <sos mergeruntime="true">
    <!-- Disable PreEmptive Analytics processing -->
    <option>disable</option>
    <!-- Send startup, shutdown, and feature messages -->
<option>sendanalytics</option>
    <!-- Do not send tamper messages -->
    <option>dontsendtamper</option>
    <!-- Send shelf life messages -->
<option>sendshelflife</option>
  </sos>
```

2.6.3.19 Extended Attributes Section

Dotfuscator allows you to tag methods or assemblies with extended attributes without modifying the application source code. Extended attributes can modify existing supported custom attributes in the code, or act as new instances of supported attributes. As it processes and transforms your application, Dotfuscator treats extended attributes the same as their custom attribute counterparts.

Using a **<codetransformlist>** element, you can also map attributes, even those embedded in code, to a specific supported code transform using extended attributes. This supports attribute overloading, wherein the same set of attributes can drive multiple transforms such as Application Analytics.

Most extended attributes may be set at the method level. To identify the method, its defining type, name, and signature must be specified.

Supported attribute arguments may be specified using a **`<propertylist>`** element.

Any attribute listed in the custom attribute reference is a supported attribute.

Supported Attributes

```
<extattributes>
<extattribute name="PreEmptive.Attributes.FeatureAttribute">
  <codetransformlist>
  <codetransform name="analytics"/>
  </codetransformlist>
  <type name = "MyApplicaton.MainForm">
   <method name="Main" signature="string[]" />
  </type>
  <propertylist>
   <property name="Name" value="Execute"/>
   <property name="ActivationStatusSinkElement" value="field"/>
   <property name="ActivationStatusSinkName" value="activated"/>
  </propertylist>
</extattribute>
</extattributes>
```

Extended runtime attributes such as **`ApplicationAttribute`**, **`BinaryAttribute`**, and **`BusinessAttribute`** may be set at the assembly level. To identify the assembly, its name must be provided in an assembly element:

Extended Runtime Attributes

```
<extattributes>
  <extattribute name="PreEmptive.Attributes.ApplicationAttribute">
    <codetransformlist>
      <codetransform name="sosruntime" />
    </codetransformlist>
    <assembly>
      <file dir="${configdir}" name="MyApp.exe" />
    </assembly>
    <propertylist>
      <property name="Version" value="" />
      <property name="Name" value="" />
      <property name="ApplicationType" value="" />
      <property name="Guid" value="00000000-0000-0000-0000-000000000000" />
    </propertylist>
  </extattribute>
</extattributes>
```

2.6.3.20 SmartObfuscation Section

Smart Obfuscation allows Dotfuscator to auto detect elements that cannot be renamed or removed based on specific rules for the application type. Smart Obfuscation is turned on by default, and in most cases should be left on. It can be turned off by setting an option in this section in cases where the user believes that aggressive obfuscation will not hurt the application.

Smart Obfuscation includes a reporting facility and this section allows you to configure the verbosity of the report. Allowed values for the verbosity attribute are **`all`**, **`warningsonly`**, and **`none`**. The default value is **`all`**.

The Smart Obfuscation report can optionally be written to disk. Dotfuscator automatically renames an existing Smart Obfuscation report with the same name before overwriting it with a new version. If you do not want Dotfuscator to rename the existing Smart Obfuscation report before overwriting, set the attribute **`overwrite="true"`**.

The XML format of the Smart Obfuscation report is discussed in the Smart Obfuscation section.

SmartObfuscation

```
<smartobfuscation>
  <!-- Skip smart obfuscation, ignoring rest of section -->
  <option>disable</option>
  <smartobfuscationreport verbosity="all" overwrite="true">
    <!-- Specifying a destination report file is optional -->
    <file dir="c:\myproject" name="smartobfuscation.xml"/>
  </smartobfuscationreport>
</smartobfuscation>
```

2.6.3.21 A Note about XML Configuration Files

Dotfuscator uses XML formatted documents for the configuration and mapping files. When loaded, these documents are validated according to the **Document Type Definitions (DTDs)** specified in the **`doctype`**. In order to perform the validation, Dotfuscator must be able to access the relevant **DTD**.

Dotfuscator takes the following steps to locate **DTDs**:

1. If the **DTD URI** specifies a local file, Dotfuscator searches for it in the indicated location. If it is not found, an error occurs.
2. If the **DTD URI** specifies a web resource, Dotfuscator first searches its cache for a file with the same name as that specified in the **URI**. Dotfuscator keeps its cache in the **%ProgramData%\PreEmptive Solutions\Common directory**.
3. If not found, Dotfuscator accesses the **URI** to obtain the **DTD**. If found, Dotfuscator caches the **DTD** so subsequent requests will not need to access the network. If the **DTD** is not found, or if Dotfuscator is unable to retrieve it from the network, an error occurs.

2.6.3.22 Custom Rules Reference

Dotfuscator allows you to customize obfuscation rules for your application. Both Inclusion Rules and Exclusion Rules provide a dynamic way to fine tune the renaming, control flow obfuscation, string encryption, and pruning of the input assemblies. These rules are applied *in addition to* rules implied by global options such as library and they are logically **OR**-ed together.

In this section

Exclusion Rules

Inclusion Rules

2.6.3.22.1 Exclusion Rules

The *exclude list* section provides a dynamic way to fine tune the renaming and control flow obfuscation of the input assemblies. The user specifies a list of rules that are applied at runtime. If a rule selects a given class, method, or field, then that item is not renamed or is excluded from control flow obfuscation.

These rules are applied *in addition to* rules implied by global options such as library.

Rules are logically **OR**-ed together.

Regular Expressions (REs) may be used to select namespaces, types, methods or fields. The optional **regex** attribute is used for this purpose. The default value of **regex** is false. If **regex** is true then the value of the **name** attribute is interpreted as a regular expression; if it is false, the name is interpreted literally. This is important since regular expressions assign special meaning to certain characters, such as the period. Here are some examples of simple, regular expressions:

Here are some examples of simple, regular expressions:

```
.*                  Matches anything
MyLibrar.           Matches MyLibrary, MyLibrari, etc.
My[\.]Test[\.]I.*   Matches My.Test.Int1,My.Test.Internal, etc.
Get.*               Matches GetInt, GetValue, etc.
Get*                Matches Ge,Get,Gett,Gettt, etc.
```

Please refer to the .NET Framework documentation for a full description of the regular expression syntax.

2.6.3.22.1.1 Excluding Namespaces

This option excludes all types and their members in a given namespace. You can use a regular expression to specify the namespace.

Regular Expression

```
<namespace name="My.Excluded.Namespace"/>
```

2.6.3.22.1.2 Excluding Types

This option excludes a type by name or by attribute specifier. You can use a regular expression to specify the type name. Type names must be fully qualified names. Inner (nested) classes are specified by using the '**/**' as a delimiter between outer and inner class. For example:

Inner (Nested) Classes

```
<type name="Library.Class1/NestedClass"/>
```

Attribute specifiers are selected or deselected with the **`speclist`** attribute. The **`speclist`** attribute is a comma-separated list of legal attribute specifiers for types. The legal values are:

Legal Values

```
abstract
interface
nestedassembly
nestedfamily
nestedfamorassem
nestedprivate
nestedpublic
notpublic
public
sealed
serializable
enum
```

A '**-**' preceding an attribute specifier negates the rule (*i.e.* it excludes all classes that do not have the specified attribute). A '**+**' may be specified but is not required. The rules implied in this list are logically **`AND`**-ed together (that is, the set of excluded types is the intersection of all types that match each rule.). For instance, the following rule excludes any type that is public AND sealed.

Exclude Public and Sealed Type Rule

```
<type name=".*" speclist="+public,+sealed" regex="true"/>
```

The **`<type>`** element may also be used to select a type in order to specify rules for field, method, property, and event exclusion within it. This allows members to be excluded while not excluding their owning type. The optional **`excludetype`** attribute is used for this purpose. If not specified, the default value is **`true`**, meaning that the type name will be excluded from renaming or control flow obfuscation.

Specify Rules for Field, Method, Property, and Event Exclusion

```
<type name="MyCo.Test.MyOtherTest" excludetype="false">
<!-- methods and fields excluded here -->
...
</type>
```

If a Type rule contains no Property or Event rules, then all property and event names in that excluded type are preserved. If a Type rule contains one or more Property rules, then only those property names will be preserved and all others will be removed. If a Type rule contains one or more Event rules, then only those event names will be preserved and all others will be removed.

Remember: If a type is **not** excluded and the library option is not set, then Dotfuscator removes property and event names.

Applies only to Renaming:

Type rules can be applied to entire inheritance hierarchies by specifying the **`applytoderivedtypes`** attribute. Setting the value of this attribute to **`true`** will apply the type rule and any Method, Field, Property, Event, Custom Attribute, or Supertype rules that it contains to the selected type and all types that derive from it. If not specified, the default value is **`false`**, meaning that the type rule will only be applied to the specified type.

2.6.3.22.1.3 Excluding Methods

Methods may be excluded by first selecting the type using the **`<type>`** element, then providing a rule for selecting methods to exclude. Methods may be excluded by name and attribute specifier, as well as by signature. Allowed attribute specifiers are:

Allowed attribute specifiers are:

```
abstract
assembly
family
familyorassembly
final
private
public
static
virtual
```

If the attribute specifier is not set explicitly, then the **`speclist`** attribute will not be used as a matching criterion. The following example selects all public instance methods beginning with Set:

The following example selects all public instance methods beginning with Set:

```
<method regex="true" name="Set.*" speclist="+public,-static"/>
```

Method signatures are specified using the **`signature`** attribute. A signature specifies both the return type and the parameter types of the method:

Return Types and Parameter Types

```
signature=""  <!-- empty signature -->

signature="string(int,MyClass,MyClass[])"
```

If the signature is not set explicitly, then the method signature will not be used as a matching criterion.

The following example selects a method by signature:

Method by Signature

```
<method name="DoIt" signature="string(int, System.Console,
System.Collection.ICollection, float[])"/>
```

Global methods may be specified by using a special type selector with the name "**`Module:mod_name`**" where **`mod_name`** is the name of the module containing the global method.

2.6.3.22.1.4 Excluding Fields

Field exclusion is valid for Renaming only. Fields may be excluded by first selecting the type using the **`<type>`** element, then providing a rule for selecting fields to exclude. Fields may also be excluded by name and attribute specifier. Allowed attribute specifiers are:

Allowed Attribute Specifiers are:

```
public
private
static
assembly
family
familyandassembly
familyorassembly
notserialized
```

If the attribute specifier is not set explicitly, then field attribute will not be used as a matching criterion. The following example selects all static fields starting with "**`ENUM_`**":

All Static Fields Starting with "ENUM":

```
<field regex="true" name="ENUM_.*" speclist="+static"/>
```

Field signatures are specified using the **`signature`** attribute. A signature specifies the type of the field:

A signature specifies the type of the field:

```
signature=""  <!-- empty signature -->

signature="int"
```

If the signature is not set explicitly, then the field type will not be used as a matching criterion.

Global fields may be specified by using a special type selector with the name **`Module:mod_name`** where **`mod_name`** is the name of the module containing the global field.

2.6.3.22.1.5 Excluding Properties

Property exclusion is valid for Renaming only. Property rules are qualified by type rules, so they appear in the rules view as children of type nodes. A property rule will select all properties (in all types matched by the parent type rule) that match your criteria. Supported matching criteria include property name and property attributes. Allowed attribute specifiers are:

Allowed Attribute Specifiers are:

```
public
private
static
assembly
family
familyandassembly
familyorassembly
```

If the attribute specifier is not set explicitly, then the property attribute will not be used as a matching criterion. The following example selects all properties starting with "Sample":

The following example selects all properties starting with "Sample":

```
<propertymember regex="true" name="Sample.*"/>
```

Property signatures are specified using the **`signature`** attribute. A signature specifies the type of the property:

A signature specifies the type of the property:

```
signature=""  <!-- empty signature -->

signature="int"
```

If the signature is not set explicitly, then the property type will not be used as a matching criterion.

Global properties may be specified by using a special type selector with the name **`Module:mod_name`** where **`mod_name`** is the name of the module containing the global property.

2.6.3.22.1.6 Excluding Events

Event exclusion is valid for Renaming only. Event rules are qualified by type rules, so they appear in the rules view as children of type nodes. An event rule will select all events (in all types matched by the parent type rule) that match your criteria. Supported matching criteria include event name and event attributes. Allowed attribute specifiers are:

Allowed attribute specifiers are:

```
public
private
static
assembly
family
familyandassembly
familyorassembly
```

If the attribute specifier is not set explicitly, then the event attribute will not be used as a matching criterion. The following example selects all events starting with "On":

The following example selects all events starting with "On":

```
<eventmember regex="true" name="On.*"/>
```

Global events may be specified by using a special type selector with the name **`Module:mod_name`** where **`mod_name`** is the name of the module containing the global event.

2.6.3.22.1.7 Excluding By Custom Attribute

Types, methods, fields, and properties may be selectively excluded by custom attribute. A custom attribute rule selects an item (type, method, field or property) based on matching against the names of custom attributes that annotate the item. One or more custom attribute rules may be nested inside any rule that selects types, methods, fields, or properties.

A type, method, field, or property rule may have multiple custom attribute rules associated with it. In this case, an item is selected if at least one of the custom attribute rules selects it.

The following example selects all types that are annotated with either **`MyCustomAttribute`** or **`MyOtherCustomAttribute`**:

Types Annotated with MyCustomAttribute or MyOther CustomAttribute:

```
<type name=".*" excludetype="false" regex="true>
   <customattribute name="MyCustomAttribute"/>
...<customattribute name="MyOtherCustomAttribute"/>
</type>
```

Custom attribute rules can also be written using regular expressions to match custom attribute names. The following example is another way to select all types annotated with either **`MyCustomAttribute`** or **`MyOtherCustomAttribute`**:

Types Annotated with MyCustomAttribute or MyOther CustomAttribute:

```
<type name=".*" excludetype="false" regex="true>
   <customattribute name="My.*CustomAttribute" regex="true"/>
</type>
```

The next example shows how to exclude all methods annotated with a custom attribute named **`MyCustomAttribute:`**

Exclude Annotated Methods

```
<type name=".*" excludetype="false" regex="true">
   <method name=".*" regex="true">
      <customattribute name="MyCustomAttribute"/>
   </method>
</type>
```

Custom attribute rules can be applied to subtypes or overriding methods and properties by specifying the **`allowinheritance`** attribute. When the value of this attribute is set to **`true`** then subtypes or overriding methods and properties with the specified custom attribute will also be excluded.

2.6.3.22.1.8 Excluding By Supertype

Types may be selectively excluded by supertype. A supertype rule selects a type based on matching against the names of types that the given type inherits from. One or more supertype rules may be nested inside any rule that selects types.

A type rule may have multiple supertype rules associated with it. In this case, an item is selected if at least one of the supertype rules selects it.

The following example selects all types that inherit from **`MySupertype`**:

All Types that Inherit from MySupertype:

```
<type name=".*" excludetype="false" regex="true">
   <supertype name="MySupertype"/>
</type>
```

Supertype rules can also be written using regular expressions to match supertype names. The following example shows how to select all types that inherit from either **`MySupertype`** or **`MyOtherSupertype`**:

All Types that Inherit from MySupertype or MyOtherSupertype:

```
<type name=".*" excludetype="false" regex="true">
   <supertype name="My.*Supertype" regex="true"/>
</type>
```

2.6.3.22.1.9 Excluding Assemblies

Assemblies may be excluded by name. When an assembly is excluded, all types and members within any of the assembly's modules are excluded. It makes sense to exclude an assembly when you have a scenario such as the following:

- Assembly **A** should be Dotfuscated.
- Assembly **B** should not be Dotfuscated.
- Assembly **B** depends on assembly **A**.

In other words, **A** provides services to B and no one else. You want the references to **A** embedded inside **B** to be Dotfuscated, so you include **B** in the same run as **A**, but you exclude **B** from renaming or control flow obfuscation.

Excluding Assemblies

```
<assembly>
  <file dir="c:\projects\project1\" name="ExcludedLib.dll"/>
</assembly>
```

2.6.3.22.1.10 Excluding Modules

Modules may be excluded by name. Use the assembly attribute to qualify the module to a particular assembly. When specified, the assembly name should be the logical assembly name rather than its physical file name. When a module is excluded, all its defined types and members are excluded.

Obviously, if a given module is shared among multiple assemblies, then the module will be excluded from all the assemblies.

Module Excluded from all Assemblies:

```
<module name="MyLibResource.dll" assemblyname="MyLib"/>
```

2.6.3.22.2 Inclusion Rules

The include list section provides a dynamic way to fine tune the string encryption and pruning of the input assemblies. The user specifies a list of rules that are applied at runtime. If a rule selects a given class, method, field, property, or event then that item is included for string encryption or pruning.

These rules are applied *in addition to* rules implied by global options such as library.

Rules are logically `OR`-ed together.

Regular Expressions (REs) may be used to select namespaces, types, methods, fields, properties, or events. The optional **`regex`** attribute is used for this purpose. The default value of **`regex`** is false. If **`regex`** is true then the value of the **`name`** attribute is interpreted as a regular expression; if it is false, the name is interpreted literally. This is important since regular expressions assign special meaning to certain characters, such as the period.

Here are some examples of simple regular expressions:

Regular Expressions:

```
.*                  Matches anything
MyLibrar.           Matches MyLibrary, MyLibrari, etc.
My[\.]Test[\.]I.*   Matches My.Test.Int1,My.Test.Internal, etc.
Get.*               Matches GetInt, GetValue, etc.
Get*                Matches Ge,Get,Gett,Gettt, etc.
```

Please refer to the .NET Framework documentation for a full description of the regular expression syntax.

2.6.3.22.2.1 Including Namespaces

This option includes all types and their methods in a given namespace. You can use a regular expression to specify the namespace.

Regular Expression:

```
<namespace name="My.Included.Namespace"/>
```

2.6.3.22.2.2 Including Types

This option includes a type by name or by attribute specifier. You can use a regular expression to specify the type name.

Type names should be fully qualified names.

Inner (nested) classes are specified by using the '**/**' as a delimiter between outer and inner class. For example:

Inner (Nested) Classes

```
<type name="Library.Class1/NestedClass"/>
```

Attribute specifiers are selected or deselected with the **`speclist`** attribute. The **`speclist`** attribute is a comma-separated list of legal attribute specifiers for types. The legal values are:

Attribute Specifiers:

```
abstract
interface
nestedassembly
nestedfamily
nestedfamorassem
nestedprivate
nestedpublic
notpublic
public
sealed
serializable
enum
```

A '**-**' preceding an attribute specifier negates the rule (*i.e.* it includes all classes that do not have the specified attribute). A '**+**' may be specified but is not required. The rules implied in this list are logically **AND**-ed together (that is, the set of included types is the intersection of all types that match each rule.). For instance, the following rule includes all methods within any type that is public AND sealed.

Include Method with Public and Sealed Types:

```
<type name=".*" speclist="+public,+sealed" regex="true"/>
```

The **<type>** element may also be used to select a type in order to specify rules for individual method inclusion within it. This allows string encryption in some methods of a type, while not in others. Note the **<type>** element's **excludetype** attribute is not used in the context of string encryption inclusions.

Allow String Encryption in Some Methods of Type:

```
<type name="MyCo.Test.MyOtherTest">
<!-- individual methods included here -->
...
</type>
```

If a **<type>** element contains no nested **<method>** elements, then all methods are selected for inclusion. This is in contrast to an exclusion rule.

Applies only to Pruning:

Type rules can be applied to entire inheritance hierarchies by specifying the **applytoderivedtypes** attribute. Setting the value of this attribute to **true** will apply the type rule and any Method, Field, Property, Event, Custom Attribute, or Supertype rules that it contains to the selected type and all types that derive from it. If not specified, the default value is **false**, meaning that the type rule will only be applied to the specified type.

2.6.3.22.2.3 Including Methods

Methods may be included by first selecting the type using the **`<type>`** element, then providing a rule for selecting methods to include. Methods may be included by name and attribute specifier (as explained in the type section above), as well as by signature. Allowed attribute specifiers are:

Allowed attribute specifiers are:

```
abstract
assembly
family
familyorassembly
final
private
public
static
virtual
```

If the attribute specifier is not set explicitly, then the **`speclist`** attribute is not used as a matching criterion.

The following example selects all public instance methods beginning with **`Set`**:

All Public Instance Methods Beginning with Set:

```
<method regex="true" name="Set.*" speclist="+public,-static"/>
```

Method signatures are specified using the **`signature`** attribute. A signature specifies both the return type and the parameter types of the method:

Return Type and Parameter Types of the Method:

```
signature=""  <!-- empty parameter list -->

signature="string(int,MyClass,MyClass[])"
```

If the signature is not set explicitly, then the method signature is not used as a matching criterion.

The following example selects a method by signature:

Select Method by Signature:

```
<method name="DoIt" signature="string(int, System.Console,
System.Collection.ICollection, float[])"/>
```

Global methods may be specified by using a special type selector with the name **`Module:mod_name`** where **`mod_name`** is the name of the module containing the global method.

2.6.3.22.2.4 Including Fields

Field inclusion is only valid for pruning triggers and pruning conditional includes.

Fields may be selected by first selecting the type using the **`<type>`** element, then providing a rule for selecting fields. Fields may be selected by name and attribute specifier (as explained in the type section above). Allowed attribute specifiers are:

Allowed attribute specifiers are:

```
public
private
static
assembly
family
familyandassembly
familyorassembly
notserialized
```

If the attribute specifier is not set explicitly, then field attribute will not be used at all as a matching criterion.

The following example selects all static fields starting with "**`ENUM_`**":

Static Fields Starting with "ENUM_":

```
<field regex="true" name="ENUM_.*" speclist="+static"/>
```

Global fields may be specified by using a special type selector with the name **`Module:mod_name`** where **`mod_name`** is the name of the module containing the global field.

2.6.3.22.2.5 Including Properties

Property inclusion is valid for Pruning only. Property rules are qualified by type rules, so they appear in the rules view as children of type nodes. A property rule will select all properties (in all types matched by the parent type rule) that match your criteria. Supported matching criteria include property name and property attributes. Allowed attribute specifiers are:

Allowed attribute specifiers are:

```
public
private
static
assembly
family
familyandassembly
familyorassembly
```

If the attribute specifier is not set explicitly, then the property attribute will not be used as a matching criterion.

The following example selects all properties starting with "**`Sample`** ":

Properties Starting with "Sample":

```
<property regex="true" name="Sample.*"/>
```

Property signatures are specified using the **`signature`** attribute. A signature specifies the type of the property:

Signature Specifies the Type of Property:

```
signature=""  <!-- empty signature -->

signature="int"
```

If the signature is not set explicitly, then the property type will not be used as a matching criterion.

Global properties may be specified by using a special type selector with the name **`Module:mod_name`** where **`mod_name`** is the name of the module containing the global property.

2.6.3.22.2.6 Including Events

Event inclusion is valid for Pruning only. Event rules are qualified by type rules, so they appear in the rules view as children of type nodes. An event rule will select all events (in all types matched by the parent type rule) that match your criteria. Supported matching criteria include event name and event attributes. Allowed attribute specifiers are:

Allowed attribute specifiers are:

```
public
private
static
assembly
family
familyandassembly
familyorassembly
```

If the attribute specifier is not set explicitly, then the event attribute will not be used as a matching criterion.

The following example selects all events starting with "**`On`**":

Events Starting with "On":

```
<event regex="true" name="On.*"/>
```

Global events may be specified by using a special type selector with the name **`Module:mod_name`** where **`mod_name`** is the name of the module containing the global event.

2.6.3.22.2.7 Including By Custom Attribute

Types and methods may be selectively included by custom attribute. A custom attribute rule selects an item (type or method) based on matching against the names of custom attributes that annotate the item. One or more custom attribute rules may be nested inside any rule that selects types or methods.

A type or method rule may have multiple custom attribute rules associated with it. In this case, an item is selected if at least one of the custom attribute rules selects it.

The following example selects all types that are annotated with either **`MyCustomAttribute`** or **`MyOtherCustomAttribute`**:

Types Annotated with MyCustomAttribute or MyOtherCustomAttribute:

```
<type name=".*" excludetype="false" regex="true>
   <customattribute name="MyCustomAttribute"/>
...<customattribute name="MyOtherCustomAttribute"/>
</type>
```

Custom attribute rules can also be written using regular expressions to match custom attribute names. The following example is another way to select all types annotated with either **`MyCustomAttribute`** or **`MyOtherCustomAttribute`**:

Selecting Types Annotated with MyCustomAttribute or MyOtherCustomAttribute:

```
<type name=".*" excludetype="false" regex="true>
   <customattribute name="My.*CustomAttribute" regex="true"/>
</type>
```

The next example shows how to include all methods annotated with a custom attribute named **`MyCustomAttribute`**:

Including Methods Annotated with MyCustomAttribute:

```
<type name=".*" excludetype="false" regex="true">
   <method name=".*" regex="true">
      <customattribute name="MyCustomAttribute"/>
   </method>
</type>
```

Custom attribute rules can be applied to subtypes or overriding methods and properties by specifying the **`allowinheritance`** attribute. When the value of this attribute is set to **`true`** then subtypes or overriding methods and properties with the specified custom attribute will also be excluded.

2.6.3.22.2.8 Including By Supertype

Types may be selectively included by supertype. A supertype rule selects a type based on matching against the names of types that the given type inherits from. One or more supertype rules may be nested inside any rule that selects types.

A type rule may have multiple supertype rules associated with it. In this case, an item is selected if at least one of the supertype rules selects it.

The following example selects all types that inherit from **`MySupertype`**:

Selecting Types that Inherit from MySuperType:

```
<type name=".*" excludetype="false" regex="true">
   <supertype name="MySupertype"/>
</type>
```

Supertype rules can also be written using regular expressions to match supertype names. The following example shows how to select all types that inherit from either **`MySupertype`** or **`MyOtherSupertype`**:

Select Types that Inherit from MySupertype or MyOtherSupertype:

```
<type name=".*" excludetype="false" regex="true">
   <supertype name="My.*Supertype" regex="true"/>
</type>
```

2.6.3.22.2.9 Including Assemblies

Assemblies may be included by name. When an assembly is included, all types and methods within any of the assembly’s modules are included.

Including Assemblies

```
<assembly>
 <file dir="c:\projects\project1\" name="IncludedLib.dll"/>
</assembly>
```

2.6.3.22.2.10 Including Modules

Modules may be included by name. Use the assembly attribute to qualify the module to a particular assembly. When specified, the assembly name should be the logical assembly name rather than its physical file name. When a module is included, all its defined types and members are included.

Obviously, if a given module is shared among multiple assemblies, then the module will be included for all the assemblies.

Including Modules

```
<module name="MyLibResource.dll" assemblyname="MyLib"/>
```

2.6.3.23 dotfuscator_v2.3.dtd

The dotfuscator_v2.3.dtd DTD describes the format of the configuration file produced by Dotfuscator. A copy is available at www.preemptive.com/dotfuscatorresources/dtd/dotfuscator_v2.3.dtd.

2.6.4 Custom Attribute Reference

This section contains reference documentation of the custom attributes and public classes used for instrumentation. These classes are defined in PreEmptive.Attributes.dll, installed by default into the Dotfuscator installation folder.

2.6.4.1 PreEmptive.Attributes

Classes

ApplicationAttribute

BinaryAttribute

BusinessAttribute

ExceptionTrackAttribute

FeatureAttribute

InsertShelfLifeAttribute

InsertSignOfLifeAttribute

InsertTamperCheckAttribute

PerformanceProbeAttribute

SetupAttribute

SystemProfileAttribute

TeardownAttribute

Enumerations

ExceptionTypes

FeatureEventTypes

InjectionPoints

SinkElements

SourceElements

2.6.4.1.1 ApplicationAttribute

Summary

An instrumentation time custom attribute for assemblies. Dotfuscator translates this attribute into an attribute that is consumable by PreEmptive Analytics code. Values specified in this attribute will be sent in PreEmptive Analytics messages to identify the application. This includes messages for Tamper Notification and Application Analytics.

ApplicationAttribute

```
public class ApplicationAttribute : Attribute
```

Constructor Members

Name	Summary
ApplicationAttribute(String guid)	Creates an ApplicationAttribute with a unique Id. The PreEmptive Analytics code will find the name and version by reflecting on the Assembly itself.
ApplicationAttribute(String guid, String name, String version)	Creates an ApplicationAttribute with a unique Id, name, and version.
ApplicationAttribute(String guid, String name, String version, String applicationType)	Creates an ApplicationAttribute with a unique Id, name, and version, and application type.

Property Members

Name	Summary
ApplicationType : String	Gets the application type. A null value is allowed.
Guid : String	Gets the application's unique Id as a string in GUID format.
Name : String	Gets the application name. A null value is allowed.
Version : String	Gets the application version. A null value is allowed.

2.6.4.1.2 BinaryAttribute

Summary

An instrumentation time custom attribute for assemblies. Dotfuscator translates this attribute into an attribute that is consumable by the PreEmptive Analytics code. Values specified in this attribute will be sent in PreEmptive Analytics messages to identify specific binaries (assemblies) that make up an application.

BinaryAttribute

```
public class BinaryAttribute : Attribute
```

Constructor Members

Name	Summary
BinaryAttribute(String guid)	Creates a BinaryAttribute with a unique Id.

Property Members

Name	Summary
Guid : String	Gets the binary's unique Id as a string in GUID format.

2.6.4.1.3 BusinessAttribute

Summary

An instrumentation time custom attribute for assemblies. Dotfuscator translates this attribute into an attribute that is consumable by the PreEmptive Analytics code. Values specified in this attribute will be sent in PreEmptive Analytics messages to identify business information.

Business Attribute

```
public class BusinessAttribute : Attribute
```

Constructor Members

Name	Summary
BusinessAttribute(String companyKey)	Creates a BusinessAttribute with a given CompanyKey.
BusinessAttribute (String companyKey, String companyName)	Creates a BusinessAttribute with a given CompanyKey and CompanyName

Property Members

Name	Summary
CompanyKey : String	Gets the company key. The company key is a token provided to each company upon registration with the service. The key may not be null or empty.
CompanyName : String	Gets the company name. A null value is allowed.

2.6.4.1.4 ExceptionTrackAttribute

Summary

The ExceptionTrackAttribute is an instrumentation time attribute for both assemblies and methods. Dotfuscator will insert exception tracking code into any assembly or method with this attribute. At runtime, the exception tracking code can detect caught, thrown, or unhandled exceptions. Multiple ExceptionTrackAttributes may be defined on an assembly or method to combine these detections. Once detected, the exception tracking code can collect details from the user and report the detected exception to a PreEmptive Analytics endpoint. A user can explicitly allow an exception report to be sent even if he or she has previously opted out of sending PreEmptive Analytics messages, and can provide comment and contact information to be sent along with the report. In addition, the developer can specify a custom action be taken when an exception is detected. In order to send Exception Tracking report messages, your application must contain methods marked with a Setup and Teardown attribute. Do not put the ExceptionTrackAttribute on the same method containing the Setup Attribute. Methods with this attribute must be executed after the method containing the Setup Attribute.

Exception Notification

A **`SinkElement`** may be used to specify a method to call, or a property or field to set in the application when an exception is detected. Unlike other **`SinkElement`** types which are always called when a check is performed, the **`ExceptionNotificationSinkElement`** is only called when an exception of the configured type is detected within the attributed assembly or method.

If the **`SinkElement`** is a property, the property's type must be **`Exception`** and its value will be set to the exception that was detected when detection occurs. If using method-level exception tracking, the property should be writable and accessible from the attributed method. If the property is not static, it must be defined in the same class as the attributed method, and the attributed method must not be static. If using assembly-level exception tracking, the property must be public, static, and writable. The name of the property must be specified in **`ExceptionNotificationSinkName`**. The type defining the property should be specified in **`ExceptionNotificationSinkOwner`**. If using method-level exception tracking, this can be left unset; in that case, Dotfuscator will look for the property on the type defining the attributed method. **`ExceptionNotificationSinkOwner`** must be set when using assembly-level exception tracking.

If the **`SinkElement`** is a field, the field's type must be **`Exception`** and its value will be set to the exception that was detected when detection occurs. If using method-level exception tracking, the field should be accessible from the attributed method. If the field is not static, it must be defined on the same class as the attributed method, and the attributed method must not be static. If using assembly-level exception tracking, the field must be public and static. The name of the field must be specified in **`ExceptionNotificationSinkName`**. The type defining the field should be specified in **`ExceptionNotificationSinkOwner`**. If using method-level exception tracking, this can be left unset; in that case, Dotfuscator will look for the field on the type defining the attributed method. **`ExceptionNotificationSinkOwner`** must be set when using assembly-level exception tracking.

If the **`SinkElement`** is a method, the method will be called with a single parameter of type **`Exception`** when detection occurs. If using method-level exception tracking, the target method should be accessible from the attributed method. If the target method is not static, it must be defined in the same class as the attributed method, and the attributed method must not be static. If using assembly-level exception tracking, the method must be public and static. The name of the target method must be specified in **`ExceptionNotificationSinkName`**. The type defining the target method should be specified in **`ExceptionNotificationSinkOwner`**. If using method-level exception tracking, this can be left unset; in that case, Dotfuscator will look for the target method on the type defining the attributed method. **`ExceptionNotificationSinkOwner`** must be set when using assembly-level exception tracking.

If the **`SinkElement`** is a method argument, the named argument should be a Delegate type and the Delegate must have this signature: `void( System.Exception )`. The Delegate will be invoked with the Exception argument set to the value of the detected exception when detection occurs. The name of the method argument must be specified in **`ExceptionNotificationSinkName`**. **`ExceptionNotificationSinkOwner`** is unused.

If the **`SinkElement`** is a Delegate, it should be a field of Delegate type. The Delegate will be invoked with the argument of type **`Exception`** set to the exception that was detected when detection occurs. If using method-level exception tracking, the field should be accessible from the attributed method. If the field is not static, it must be defined on the same class as the attributed method, and the attributed method must not be static. If using assembly-level exception tracking, the field must be public and static. The name of the field must be specified in **`ExceptionNotificationSinkName`**. **`ExceptionNotificationSinkOwner`** is unused.

If the **`ExceptionNotificationSinkElement`** is set to DefaultAction or None, Dotfuscator will not inject any code to react to the detected exception. Dotfuscator will still generate code to send the appropriate message to a PreEmptive Analytics Endpoing if configured to do so via the **`SendReport`** attribute property.

Dotfuscator will remove this custom attribute from the metadata after instrumentation.

ExceptionTrackAttribute

```
public class ExceptionTrackAttribute : Attribute
```

Constructor Members

Name	Summary
ExceptionTrackAttribute()	Initializes a new instance of the class.

Property Members

Name	**Summary**
ExceptionNotificationSinkElement : SinkElements	Indicates whether and how to take action upon detection of an exception of the configured ExceptionType. A sink element may be a writable field or settable property of type `Exception`, a void(System.Exception) method to call, or a Delegate to invoke. To use this property, `ExceptionNotificationSinkName` must also be set. If using method-level exception tracking and `ExceptionNotificationSinkElement` is a field, method, or property, `ExceptionNotificationSinkOwner` must also be set unless the field, method or property is defined on the same class as the attributed method. If using assembly-level exception tracking, `ExceptionNotificationSinkOwner` must always also be set. If the `SinkElement` is set to "None" or "DefaultAction", Dotfuscator will not inject any code to react to the detected exception. If the `SendReport` property is set to `true`, code to send an exception report will still be injected regardless of whether this property has been set to "None" or "DefaultAction".
ExceptionNotificationSinkName : String	The name of the property, field, method to set or call when an exception of the configured type is detected. If using assembly-level exception tracking, this property, field, or method must be public, static, and writable. If this property is set, `ExpirationNotificationSinkElement` is required to be set as well. If this property is not set, Dotfuscator will not inject any code to react to the detected exception. If the `SendReport` property is set to `true`, code to send an exception report will still be injected regardless of whether this property is set.
ExceptionNotificationSinkOwner : Type	If the `ExceptionNotificationSinkElement` is a field, method or property, `ExceptionNotificationSinkOwner` indicates the name of the type that defines the field, method or property. If not set, the named source element is searched for on the attributed method's type. `ExceptionNotificationSinkOwner` must be set for assembly-level exception tracking.
ExceptionType : ExceptionTypes	The type of exceptions to track with this ExceptionTrackAttribute (caught, thrown, or unhandled). The default is `Unhandled`.

ExtendedKeyMethodArguements : string	A pattern indicating which parameter names and values should be automatically added to the messages extended key data at runtime. See Automatically Sending Method Parameters as Extended Keys for details on supported patterns.
ExtendedKeySourceElement : SourceElements	Indicates how to access the extended key dictionary at runtime, at the time that the attributed method is called (e.g. a field, property, method, or method parameter). To use this property, `ExtendedKeySourceName` must also be set. If using method-level exception tracking and `ExtendedKeySourceElement` is a field, method, or property, `ExtendedKeySourceOwner` must also be set unless the field, method or property is defined on the same class as the attributed method. If using assembly-level exception tracking, `ExtendedKeySourceOwner` must always also be set.
ExtendedKeySourceName : string	The name of the property, field, method, or method parameter that will contain the extended key dictionary at runtime, at the time that the attributed method is called. If using assembly-level exception tracking, this property, field, or method must be public, static, and writable. If this property is set, `ExtendedKeySourceElement` is required to be set as well. If this property is not set, Dotfuscator will not generate code to send extended key information.
ExtendedKeySourceOwner : Type	If the `ExtendedKeySourceElement` is a field, method or property, `ExtendedKeySourceOwner` indicates the name of the type that defines the field, method or property. If not set, the named source element is searched for on the attributed method’s type. `ExtendedKeySourceOwner` must be set for assembly-level exception tracking.
PrivacyPolicyUri : string	URL to a privacy policy covering the transmission of exception reports. This property is only used if the `ReportInfoSourceElement` is set to DefaultAction. In this case, a link to the PrivacyPolicyUri is included on the built-in exception report dialog. If this property is not set, the exception report dialog will not contain a privacy policy link.
ReportInfoSourceElement : SourceElements	Indicates how to access the user-specified report information dictionary at runtime, at the time an exception is detected. The default value is None. If this property is set to None, Dotfuscator will not generate code that obtains user-specified exception report information. If this property is set to DefaultAction, Dotfuscator will generate code that displays a built-in dialog that shows the exception message

	and prompts the user for his or her address, comments, and consent to send the report. If this property is set to any other value, `ReportInfoSourceName` must also be set. If using method-level exception tracking and `ReportInfoSourceElement` is a field, method, or property, `ReportInfoSourceOwner` must also be set unless the field, method or property is defined on the same class as the attributed method. If using assembly-level exception tracking, `ReportInfoSourceOwner` must always also be set. See Collecting User-specified Exception Report Information for a description of the key-value pairs recognized in the user-specified report information dictionary.
ReportInfoSourceName : string	The name of the property, field, method, or method parameter that will contain the user-specified report information dictionary at runtime, at the time that the attributed method is called. If using assembly-level exception tracking, this property, field, or method must be public, static, and writable. If this property is set, `ReportInfoSourceElement` is required to be set as well. If this property is not set, Dotfuscator will not generate code that obtains user-specified exception report information.
ReportInfoSourceOwner : Type	If the `ReportInfoSourceElement` is a field, method or property, `ReportInfoSourceOwner` indicates the name of the type that defines the field, method or property. If not set, the named source element is searched for on the attributed method's type. `ReportInfoSourceOwner` must be set for assembly-level exception tracking.
SendReport : bool	Whether or not to send an exception report when an exception is tracked. The default value is `true`. If this property is set to `false`, the ReportInfoSource properties will be ignored. A custom action (via the ExceptionNotifiationSink properties) will always be taken if one has been configured, regardless of the value of this property.

2.6.4.1.4.1 ExceptionTypes

Summary

Types of exceptions to track.

Exception Types

```
public enum ExceptionTypes
```

Enumeration Members

Field	Summary
Unhandled	Track unhandled exceptions.
Caught	Track exceptions right after they enter a 'catch' block.
Thrown	Track exceptions right before being thrown by a 'throw' statement.

2.6.4.1.5 FeatureAttribute

Summary

A FeatureAttribute is an instrumentation time custom attribute for Application Analytics processing. Dotfuscator will insert code into the attributed method that sends PreEmptive Analytics feature messages.

Dotfuscator will remove this custom attribute from the metadata after instrumentation.

Feature Event Types

Features can be defined as one time only events (Ticks), or they can be defined with separate Stop and Start events. The event type you choose affects the type of code Dotfuscator generates.

The Tick event results in one PreEmptive Analytics feature message being sent when the attributed method executes. For Start/Stop events, two separate messages are sent: one for start and one for stop.

To use Start/Stop events, two FeatureAttributes are required (which can go on the same method if desired). Code generated for the Start event is added at the beginning of the attributed method, while code for the Stop event is added to the end of the method.

Feature Attribute

```
public class FeatureAttribute : Attribute
```

Constructor Members

Name	Summary
FeatureAttribute(String featureName)	Creates a new `FeatureAttribute` representing the named feature. The name of the feature should match the name of a feature or feature set specified for the product.

Property Members

Name	Summary
ExtendedKeyMethodArguements : string	A pattern indicating which parameter names and values should be automatically added to the messages' extended key data at runtime. See Automatically Sending Method Parameters as Extended Keys for details on supported patterns.
ExtendedKeySourceElement : SourceElements	Indicates how to access the extended key dictionary at runtime, at the time that the attributed method is called (e.g. a field, property, method, or method parameter). To use this property, `ExtendedKeySourceName` must also be set. If `ExtendedKeySourceElement` is a field, method, or property, `ExtendedKeySourceOwner` must also be set unless the field, method or property is defined on the same class as the attributed method.

ExtendedKeySourceName : string	The name of the property, field, method, or method parameter that will contain the extended key dictionary at runtime, at the time that the attributed method is called. If this property is set, `ExtendedKeySourceElement` is required to be set as well. If this property is not set, Dotfuscator will not generate code to send extended key information.
ExtendedKeySourceOwner : Type	If the `ExtendedKeySourceElement` is a field, method or property, `ExtendedKeySourceOwner` indicates the name of the type that defines the field, method or property. If not set, the named source element is searched for on the attributed method's type.
EventType : FeatureEventTypes	Describes what type of feature event this is. For analytics, Dotfuscator will send different messages based on the event type. Bracketing feature usage with Start and Stop pairs allows for tracking of time spent in the feature.

2.6.4.1.5.1 FeatureEventTypes

Summary

Possible feature event types for the FeatureAttribute.

FeatureEventTypes
`public enum FeatureEventTypes : Attribute`

Enumeration Members

Field	Summary
Tick	This `FeatureAttribute` marks a method that represents a feature use as an instant in time.
Start	This `FeatureAttribute` marks a method that represents the beginning of a Feature.
Stop	This `FeatureAttribute` marks a method that represents the end of a Feature.

2.6.4.1.6 InsertShelfLifeAttribute

Summary

The InsertShelfLifeAttribute is an instrumentation time attribute for methods. Dotfuscator will insert shelf life code into any method with this attribute. At runtime, the shelf life code can send a warning or expiration message if the application has expired or is about to expire. If the warning period is entered, then a user defined action can be executed. Upon expiration, the default behavior is to exit the application, though a user-defined action can be executed instead. In order to use Shelf Life your application must contain methods marked with a Setup and Teardown attribute. Do not put the InsertShelfLife attribute on the same method containing the Setup Attribute. Methods with this attribute must be executed after the method containing the Setup Attribute.

Application Notification

A **`SinkElement`** may be used to specify a method to call, or a property or field to set in the application when a warning or expiration event occurs.

If the **`SinkElement`** is a property, the property will be set to true if the warning period is entered or the application is expired; false if not. The property should be writable, accessible from the attributed method, and of boolean type. If the property is not static, it must be defined in the same class as the attributed method, and the attributed method must not be static. The name of the property must be specified in **`ExpirationNotificationSinkName`** or **`WarningNotificationSinkName`**. The generated code will make no attempt to catch or handle exceptions thrown from the property setter. The type defining the property should be specified in **`ExpirationNotificationSinkOwner`** or **`WarningNotificationSinkOwner`**; if either is left unset, Dotfuscator will look for the property on the type defining the attributed method.

If the **`SinkElement`** is a field, the field will be set to true if warning or expiration occurs; false if not. The field should be accessible from the attributed method, and of boolean type. If the field is not static, it must be defined on the same class as the attributed method, and the attributed method must not be static. The name of the field must be specified in **`ExpirationNotificationSinkName`** or **`WarningNotificationSinkName`** . The type defining the field should be specified in **`ExpirationNotificationSinkOwner`** or **`WarningNotificationSinkOwner`**; if either is left unset, Dotfuscator will look for the field on the type defining the attributed method.

If the **`SinkElement`** is a method, the method will be called with a single Boolean parameter set to true if the warning period is entered or the application is expired; false if not. The target method should be accessible from the attributed method, and must take one parameter of boolean type. If the target method is not static, it must be defined in the same class as the attributed method, and the attributed method must not be static. The name of the target method must be specified in **`ExpirationNotificationSinkName`** or **`WarningNotificationSinkName`**. The generated code will make no attempt to catch or handle exceptions thrown from the target method. The type defining the target method should be specified in **`ExpirationNotificationSinkOwner`** or **`WarningNotificationSinkOwner`**; if either is left unset, Dotfuscator will look for the target method on the type defining the attributed method.

If the **`SinkElement`** is a method argument, the named argument should be a Delegate type and the Delegate must have this signature: `void( bool )`. The Delegate will be invoked with the boolean argument set to true if the warning period is entered or the application is expired; false if not. The generated code will make no attempt to catch or handle exceptions thrown from the invoked Delegate. The name of the method argument must be specified in **`ApplicationNotificationSinkName`**. **`ApplicationNotificationSinkOwner`** is unused.

If the **`SinkElement`** is a Delegate, it should be a field of Delegate type. The Delegate will be invoked with the boolean argument set to true if the warning period is entered or the application is expired; false if not. If the field is not static, it must be defined on the same class as the attributed method, and the attributed method must not be static. The name of the field must be specified in **`ExpirationNotificationSinkName`** or **`WarningNotificationSinkName`**. In this case, the **`ExpirationNotificationSinkOwner`** or **`WarningNotificationSinkOwner`** are unused**.** The Delegate must have this signature: **`void( bool )`**. The generated code will make no attempt to catch or handle exceptions thrown from the invoked Delegate.

If the **`ExpirationNotificationSinkElement`** is set to DefaultAction, Dotfuscator will inject code that exits the application if it is expired. If the **`WarningNotificationSinkElement`** is set to DefaultAction, Dotfuscator will not inject any warning specific code.

If the **`SinkElement`** is set to None, Dotfuscator will not inject any code to react if the warning period is entered or the application is expired. Dotfuscator will still generate code to send the appropriate message to a PreEmptive Analytics Endpoint if configured to do so from the *Global Options* tab.

Dotfuscator will remove this custom attribute from the metadata after instrumentation.

InsertShelfLifeAttribute

```
public class InsertShelfLifeAttribute : Attribute
```

Constructor Members

Name	Summary
InsertShelfLifeAttribute()	Initializes a new instance of the class.

Property Members

Name	Summary
ActivationKeyFile	The path to the Shelf Life Activation Key file.
ExpirationDate	Indicates the date the application will expire and/or deactivate. This can be an absolute date (e.g., 2009-12-31) or a positive integer representing number of days after the Dotfuscation date that the application should expire.
ExpirationNotificationSinkElement : SinkElements	Indicates whether and how to notify the application after shelf life expiration, at the time that the attributed method is called. A sink element may be a writable boolean field, settable boolean property, a void(boolean) method to call, or a Delegate to invoke. To use this property, `ExpirationNotificationSinkName` must also be set. If `ExpirationNotificationSinkElement` is a field, property, or method, the `ExpirationNotificationSinkOwner` should also be set if it is not in the same class as the attributed method. If the `SinkElement` is set to “DefaultAction”, Dotfuscator will inject code that exits the application if shelf life expiration is detected.
ExpirationNotificationSinkName : String	The name of the property, field, method to set or call after a shelf life expiration check. If this property is set, `ExpirationNotificationSinkElement` is required to be set as well. If this property is not set, Dotfuscator will not generate code that notifies the application of expiration.
ExpirationNotificationSinkOwner : Type	`ExpirationNotificationSinkOwner` indicates the name of the type that defines the `ExpirationNotificationSink` method, field, property, argument, or delegate. If not set, the named sink element is searched for on the attributed method’s type.
ExtendedKeyMethodArguements : string	A pattern indicating which parameter names and values should be automatically added to the messages extended key data at runtime. See Automatically Sending Method Parameters as Extended Keys for details on supported patterns.
ExtendedKeySourceElement : SourceElements	Indicates how to access the extended key dictionary at runtime, at the time that the attributed method is called (e.g. a field, property, method, or method parameter). To use this property, `ExtendedKeySourceName` must also be

	set. If `ExtendedKeySourceElement` is a field, method, or property, `ExtendedKeySourceOwner` must also be set unless the field, method or property is defined on the same class as the attributed method.
ExtendedKeySourceName : string	The name of the property, field, method, or method parameter that will contain the extended key dictionary at runtime, at the time that the attributed method is called. If this property is set, `ExtendedKeySourceElement` is required to be set as well. If this property is not set, Dotfuscator will not generate code to send extended key information.
ExtendedKeySourceOwner : Type	If the `ExtendedKeySourceElement` is a field, method or property, `ExtendedKeySourceOwner` indicates the name of the type that defines the field, method or property. If not set, the named source element is searched for on the attributed method’s type.
PrivateKeyFile	The path to a PKCS #12 Private Key file that is optionally used to provide additional validation of a Shelf Life token.
PrivateKeyFilePassword	The password protecting the Private Key file.
ShelfLifeTokenSourceElement	`ShelfLifeTokenSourceElement` indicates how to access the optional shelf life token source at runtime, at the time that the attributed method is called (e.g. a field, property, method, or method parameter). To use this property, `ShelfLifeTokenSourceName` must also be set. If `ShelfLifeTokenSourceElement` is a field, method, or property, `ShelfLifeTokenSourceOwner` must also be set unless the field, method or property is defined on the same class as the attributed method.
ShelfLifeTokenSourceName	The name of the property, field, method, or method parameter that will contain the shelf life token at runtime, at the time that the attributed method is called. If this property is set, `ShelfLifeTokenSourceElement` is required to be set as well. If this property is not set, Dotfuscator will create a shelf life token from the shelf life activation key during instrumentation.
ShelfLifeTokenSourceOwner	If the `ShelfLifeTokenSourceElement` is a field, method or property, `ShelfLifeTokenSourceOwner` indicates the name of the type that defines the field, method or property. If not set, the named source element is searched for on the attributed method’s type.
WarningDate	Indicates the date a warning of impending application expiration will occur. This can be an absolute date

	(e.g., 2009-12-31) or a positive integer representing number of days after the Dotfuscation date that the application should issue a warning.
WarningNotificationSinkElement	Indicates whether and how to notify the application of impending shelf life expiration, at the time that the attributed method is called. A sink element may be a writable boolean field, settable boolean property, a void(boolean) method to call, or a Delegate to invoke. To use this property, `WarningNotificationSinkName` must also be set. If ApplicationNotificationSinkElement is a field, property, or method, the `WarningNotificationSinkOwner` should also be set if it is not in the same class as the attributed method. If the `SinkElement` is set to "DefaultAction", Dotfuscator will not inject code to perform any additional warning behavior.
WarningNotificationSinkName	The name of the property, field, method to set or call after a shelf life warning check. If this property is set, `WarningNotificationSinkElement` is required to be set as well. If this property is not set, Dotfuscator will not inject code to perform any additional warning behavior.
WarningNotificationSinkOwner	`WarningNotificationSinkOwner` indicates the name of the type that defines the `WarningNotificationSink` method, field, property, argument, or delegate. If not set, the named sink element is searched for on the attributed method's type.

2.6.4.1.7 InsertSignofLifeAttribute

Summary

The InsertSignOfLifeAttribute sends a message each time a method instrumented with this attribute is called. At runtime, the sign of life code will send a message indicating that the application has been executed. In order to use Sign Of Life your application must contain methods marked with a Setup and Teardown attribute. Do not put the InsertSignOfLife attribute on the same method containing the Setup Attribute. Methods with this attribute must be executed after the method containing the Setup Attribute.

Dotfuscator will remove this custom attribute from the metadata after instrumentation.

InsertSignofLifeAttribute

```
public class InsertShelfLifeAttribute : Attribute
```

Constructor Members

Name	Summary
InsertSignOfLifeAttribute()	Initializes a new instance of the class.

Property Members

Name	Summary
ActivationKeyFile	The path to the Shelf Life Activation Key file.
ExtendedKeyMethodArguements : string	A pattern indicating which parameter names and values should be automatically added to the messages extended key data at runtime. See Automatically Sending Method Parameters as Extended Keys for details on supported patterns.
ExtendedKeySourceElement : SourceElements	Indicates how to access the extended key dictionary at runtime, at the time that the attributed method is called (e.g. a field, property, method, or method parameter). To use this property, `ExtendedKeySourceName` must also be set. If `ExtendedKeySourceElement` is a field, method, or property, `ExtendedKeySourceOwner` must also be set unless the field, method or property is defined on the same class as the attributed method.
ExtendedKeySourceName : string	The name of the property, field, method, or method parameter that will contain the extended key dictionary at runtime, at the time that the attributed method is called. If this property is set, `ExtendedKeySourceElement` is required to be set as well. If this property is not set, Dotfuscator will not generate code to send extended key information.
ExtendedKeySourceOwner : Type	If the `ExtendedKeySourceElement` is a field, method or property, `ExtendedKeySourceOwner` indicates the name of the type that defines the field, method or property. If not set, the named source element is searched for on the attributed method's type.

2.6.4.1.8 InsertTamperCheckAttribute

Summary

The InsertTamperCheckAttribut is an instrumentation time attribute for methods. Dotfuscator will insert tamper checking code into any method with this attribute. At runtime, the tamper checking code can send a PreEmptive Analytics tamper message if the application integrity checks fail. It can also invoke custom code in your application or invoke code that simply exits the application. If you wish to send PreEmptive Analytics tamper messages, do not put this attribute on the same method containing the Setup Attribute. Methods with this attribute must be executed after the method containing the Setup Attribute.

Application Notification

A **`SinkElement`** may be used to specify a method to call or a property or field to set in the application after the tamper check is performed. This allows the application to specify what should occur inside the application when tampering is or isn't detected. The specified field or property will be set (or the method or delegate will be invoked) all the time-- even if a tamper is not detected.

If the **`SinkElement`** is a property, the property will be set to true if tampering is detected; false if not. The property should be writable, accessible from the attributed method, and of boolean type. If the property is not static, it must be defined on the same class as the attributed method, and the attributed method must not be static. The name of the property must be specified in **`ApplicationNotificationSinkName`**. The generated code will make no attempt to catch or handle exceptions thrown from the property setter. The type defining the property should be specified in **`ApplicationNotificationSinkOwner`**; if **`ApplicationNotificationSinkOwner`** is left unset, Dotfuscator will look for the property on the type defining the attributed method.

If the **`SinkElement`** is a field, the field will be set to true if tampering is detected; false if not. The field should be accessible from the attributed method, and of boolean type. If the field is not static, it must be defined on the same class as the attributed method, and the attributed method must not be static. The name of the field must be specified in **`ApplicationNotificationSinkName`**. The type defining the field should be specified in ApplicationNotificationSinkOwner; if **`ApplicationNotificationSinkOwner`** is left unset, Dotfuscator will look for the field on the type defining the attributed method.

If the **`SinkElement`** is a method argument, the named argument should be a Delegate type and the Delegate must have this signature: `void( bool )`. The Delegate will be invoked with the boolean argument set to true if a tamper is detected; false if not. The generated code will make no attempt to catch or handle exceptions thrown from the invoked Delegate. The name of the method argument must be specified in **`ApplicationNotificationSinkName`**. **`ApplicationNotificationSinkOwner`** is unused.

If the **`SinkElement`** is a Delegate, it should be a field of Delegate type. The Delegate will be invoked with the boolean argument set to true if a tamper is detected; false if not. If the field is not static, it must be defined on the same class as the attributed method, and the attributed method must not be static. The name of the field must be specified in **`ApplicationNotificationSinkName`**. **`ApplicationNotificationSinkOwner`** is unused. The Delegate must have this signature: `void( bool )`. The generated code will make no attempt to catch or handle exceptions thrown from the invoked Delegate.

If the **`SinkElement`** is set to DefaultAction, Dotfuscator will inject code that exits the application if tampering is detected.

If the **`SinkElement`** is set to None, Dotfuscator will not inject any code to react if a tamper is detected (though Dotfuscator will still generate code to send a message to a PreEmptive Analytics Endpoint if configured to do so).

Dotfuscator will remove this custom attribute from the metadata after instrumentation.

InsertTamperCheckAttribute

```
public class InsertTamperCheckAttribute : Attribute
```

Constructor Members

Name	Summary
InsertTamperCheckAttribute()	Initializes a new instance of the class.

Property Members

Name	Summary
ApplicationNotificationSinkElement : SinkElements	Indicates whether and how to notify the application after a tamper check, at the time that the attributed method is called. A sink element may be a writable boolean field, settable boolean property, a void(boolean) method to call, or a Delegate to invoke. To use this property, `ApplicationNotificationSinkName` must also be set. If `ApplicationNotificationSinkElement` is a field, property, or method, the `ApplicationNotificationSinkOwner` should also be set if it is not in the same class as the attributed method. If the `SinkElement` is set to "DefaultAction", Dotfuscator will inject code that exits the application if tampering is detected.
ApplicationNotificationSinkName : String	The name of the property, field, method to set or call after a tamper check. If this property is set, `ApplicationNotificationSinkElement` is required to be set as well. If this property is not set, Dotfuscator will not generate code that notifies the application of a tamper.
ApplicationNotificationSinkOwner : Type	`ApplicationNotificationSinkOwner` indicates the name of the type that defines the `ApplicationNotificationSink` method, field, property, argument, or delegate. If not set, the named sink element is searched for on the attributed method's type.
ExtendedKeyMethodArguements : string	A pattern indicating which parameter names and values should be automatically added to the messages extended key data at runtime. See Automatically Sending Method Parameters as Extended Keys for details on supported patterns.
ExtendedKeySourceElement : SourceElements	Indicates how to access the extended key dictionary at runtime, at the time that the attributed method is called (e.g. a field, property, method, or method parameter). To use this property, `ExtendedKeySourceName` must also be set. If `ExtendedKeySourceElement` is a field, method, or property, `ExtendedKeySourceOwner` must also be set unless the field, method or property is defined on the same class as the attributed method.
ExtendedKeySourceName : string	The name of the property, field, method, or method parameter that will contain the extended key dictionary at runtime, at the time that the attributed method is called. If

	this property is set, `ExtendedKeySourceElement` is required to be set as well. If this property is not set, Dotfuscator will not generate code to send extended key information.
ExtendedKeySourceOwner : Type	If the `ExtendedKeySourceElement` is a field, method, or property, `ExtendedKeySourceOwner` indicates the name of the type that defines the field, method, or property. If not set, the named source element is searched for on the attributed method's type.

s

2.6.4.1.9 PerformanceProbeAttribute

Summary

The PerformanceProbeAttribute is an instrumentation time attribute for methods. Dotfuscator will insert code to generate and send a PreEmptive Analytics PerformanceProbe Message into any method with this attribute. Dotfuscator will remove this custom attribute from the metadata after instrumentation.

PerformanceProbe Attribute

```
public class PerformanceProbeAttribute : Attribute
```

Constructor Members

Name	Summary
PerformanceProbeAttribute()	Initializes a new instance of the class.

Property Members

Name	Summary
ExtendedKeyMethodArguements : string	A pattern indicating which parameter names and values should be automatically added to the messages extended key data at runtime. See Automatically Sending Method Parameters as Extended Keys for details on supported patterns.
ExtendedKeySourceElement : SourceElements	Indicates how to access the extended key dictionary at runtime, at the time that the attributed method is called (e.g. a field, property, method, or method parameter). To use this property, `ExtendedKeySourceName` must also be set. If `ExtendedKeySourceElement` is a field, method, or property, `ExtendedKeySourceOwner` must also be set unless the field, method, or property is defined on the same class as the attributed method.
ExtendedKeySourceName : string	The name of the property, field, method, or method parameter that will contain the extended key dictionary at runtime, at the time that the attributed method is called. If this property is set, `ExtendedKeySourceElement` is required to be set as well. If this property is not set, Dotfuscator will not generate code to send extended key information.
ExtendedKeySourceOwner : Type	If the `ExtendedKeySourceElement` is a field, method or property, `ExtendedKeySourceOwner` indicates the name of the type that defines the field, method or property. If not set, the named source element is searched for on the attributed method's type.
InjectionPoint : InjectionPoints	Where in the method to inject the generated code. Default to beginning.
Name : string	The name of this instance of the `PerformanceProbeAttribute`. This can be used to distinguish between performance measurements taken at different points during execution.

2.6.4.1.9.1 InjectionPoints

Summary

Places in a method to inject generated code.

Injection Points

```
public enum InjectionPoints
```

Enumeration Members

Field	Summary
Beginning	Inject code at the beginning of the method.
End	Inject code at the end of the method.

2.6.4.1.10 SetupAttribute

Summary

The SetupAttribute is an instrumentation time attribute for methods. This attribute can be used to guide Application Analytics instrumentation.

When applied to Application Analytics, Dotfuscator will insert PreEmptive Analytics initialization code into methods with this attribute. There must be one or more methods with this attribute in an assembly or application that uses PreEmptive Analytics code. At runtime, the initialization code will send PreEmptive Analytics application and session start messages when this method is called. To further configure PreEmptive Analytics message sending, the developer can optionally specify information about an application instance ID and an "opt in" flag. Dotfuscator will use this information when generating the initialization code.

To configure the endpoint or destination of the analytics messages, Dotfuscator allows the StaticEndpoint property to be set to an endpoint of your choice and will default to the commercial Runtime Intelligence portal.

Instance ID

The application instance ID (such as a serial number) is typically unique to a particular instance of the application and is determined by the application at runtime. The developer can make the instance ID available at runtime to the PreEmptive Analytics initialization code by specifying values for the **`InstanceIdSourceName`**, **`InstanceIdSourceElement`**, and (optionally) **`InstanceIdSourceOwner`** properties. These fields are used to generate code that makes the instance ID available to send in PreEmptive Analytics messages.

The **`InstanceIdSourceElement`** may be a string property, a string field, a no argument method that returns a string, or a string method argument.

If it is a property or field, it should be writable, accessible from the attributed method, and of string type. If it is a method, it should also be accessible from the attributed method. If the property, method, or field is not static, it must be defined on the same class as the attributed method, and the attributed method must not be static. The name of the property, method, or field must be specified in **`InstanceIdSourceName`**. The type defining the field, method, or property should be specified in **`InstanceIdSourceOwner`**; if **`InstanceIdSourceOwner`** is left unset, Dotfuscator will look for the property, method, or field on the type defining the attributed method.

If the **`InstanceIdSourceElement`** is a method argument, the named argument should be of type string. The name of the method argument must be specified in **`InstanceIdSourceName`**. **`InstanceIdSourceOwner`** is unused

Offline Storage of Usage Data

PreEmptive Analytics instrumented applications have the ability to store usage data in situations when network access is unavailable and then transmit the data when connectivity is restored. Usage data is stored in Isolated Storage. This behavior is enabled by default and default connectivity detection code is injected into instrumented applications. Developers can override the default behavior by changing the `OfflineStateSourceElement`. If the `OfflineStateSourceElement` value is changed to **None** then usage data will not be stored when the application is unable to connect to the network and that usage data will be dropped. Developers also have the ability to write their own network detection code and make the network connectivity state available to the PreEmptive Analytics code by specifying the applications offline state in a boolean value in the instrumented method's parameters, as the return value of a method or in a field or property. This is accomplished by setting the `OfflineStateSourceElement` property to the appropriate value and setting the `OfflineStateSourceName` and `OfflineStateSourceOwner`.

The application can also be notified of the success or failure of an attempt to store usage data to the offline storage mechanism via the `OfflineStorageResultSinkElement`. If the value is **None** then no notification will be made of the success or failure of the data storage. If the value is **DefaultAction** then if the storage mechanism is unable to store any usage data the application will exit immediately. Developers can write code to react to the success or failure of offline storage by setting the OfflineStorageResultSinkElement to the appropriate value and setting the `OfflineStorageResultSinkName` and `OfflineStorageResultSinkOwner`. The selected application code will be called with a parameter value or the boolean property or field will be set with the result of the most recent attempt to save usage data to the offline storage mechanism.

Opt In

The Opt-In flag is another runtime determined value that indicates if the user of the application has given permission for the application to send PreEmptive Analytics data. The developer can make the opt-in flag available at runtime to the PreEmptive Analytics initialization code by specifying values for the **`OptInSourceName`**, **`OptInSourceElement`**, and (optionally) **`OptInSourceOwner`** properties. These fields are used to generate code that makes the opt-in preference available to the PreEmptive Analytics message sender.

The **`OptInSourceElement`** may be a boolean property, a boolean field, a no argument method that returns a boolean, or a boolean method argument.

If it is a property or field, it should be writable, accessible from the attributed method, and of boolean type. If it is a method, it should also be accessible from the attributed method. If the property, method, or field is not static, it must be defined on the same class as the attributed method, and the attributed method must not be static. The name of the property, method, or field must be specified in **`OptInSourceName`**. The type defining the field, method, or property should be specified in OptInSourceOwner; if **`OptInSourceOwner`** is left unset, Dotfuscator will look for the property, method, or field on the type defining the attributed method.

If the **`OptInSourceElement`** is a method argument, the named argument should be of type boolean. The name of the method argument must be specified in **`OptInSourceName`**. **`OptInSourceOwner`** is unused

Dotfuscator will remove this custom attribute from the metadata after instrumentation.

SetupAttribute

```
public class SetupAttribute : Attribute
```

Constructor Members

Name	Summary
SetupAttribute()	Initializes a new instance of the class.

Property Members

Name	Summary
EndpointSourceElement : SourceElements	Indicates if the StaticEndpoint or a runtime determined endpoint will be used. Specifying `None` or `DefaultAction` will use the endpoint specified by the `StaticEndpoint` attribute.
EndpointSourceName : String	The name of the property, field, method, or method parameter that will contain the endpoint URL. If this property is set, `EndpointSourceElement` is required to be set to something other than `None` or `DefaultAction`.
EndpointSourceOwner : Type	If the `EndpointSourceElement` is a field, method or property, `EndpointSourceOwner` indicates the name of the type that defines the field, method or property. If not set, the named source element is searched for on the attributed method’s type.
ExtendedKeyMethodArguements : String	A pattern indicating which parameter names and values should be automatically added to the messages extended key data at runtime. See Automatically Sending Method Parameters as Extended Keys for details on supported patterns.
ExtendedKeySourceElement : SourceElements	Indicates how to access the extended key dictionary at runtime, at the time that the attributed method is called (e.g. a field, property, method, or method parameter). To use this property, `ExtendedKeySourceName` must also be set. If `ExtendedKeySourceElement` is a field, method, or property, `ExtendedKeySourceOwner` must also be set unless the field, method or property is defined on the same class as the attributed method.
ExtendedKeySourceName : String	The name of the property, field, method, or method parameter that will contain the extended key dictionary at runtime, at the time that the attributed method is called. If this property is set, `ExtendedKeySourceElement` is required to be set as well. If this property is not set, Dotfuscator will not generate code to send extended key information.
ExtendedKeySourceOwner : Type	If the `ExtendedKeySourceElement` is a field, method or property, `ExtendedKeySourceOwner` indicates the name of the type that defines the field, method or property. If not set, the named source element is searched for on the attributed method’s type.
InstanceIdSourceElement : SourceElements	Indicates how to access the instance ID at runtime, at the time that the setup method is called (e.g. a field, property, or method parameter). To use this property, `InstanceIdSourceName` must also be set. If `InstanceIdSourceElement` is a field or property,

	`InstanceIdSourceOwner` must also be set unless the field or property is defined on the same class as the attributed method.
InstanceIdSourceName : String	The name of the property, field, or method parameter that will contain the instance ID at runtime, at the time that the setup method is called. If this property is set, `InstanceIdSourceElement` is required to be set as well. If this property is not set, Dotfuscator will not generate code that sends an instance Id with PreEmptive Analytics messages.
InstanceIdSourceOwner : Type	If the `InstanceIdSourceElement` is a field or property, `InstanceIdSourceOwner` indicates the name of the type that defines the field or property. If not set, the named source element is searched for on the attributed method's type.
OfflineStateSourceElement : SourceElements	Indicates how to access the applications offline/online state at runtime, at the time that the setup method is called (e.g. a field, property, or method parameter). If neither **DefaultAction** nor **None** is selected then `OfflineStateSourceName` must also be set. If `OfflineStateSourceElement` is a field or property, `OfflineStateSourceOwner` must also be set unless the field or property is defined on the same class as the attributed method. If the value of this property is **DefaultAction** then application usage data will be automatically stored by the offline storage mechanism (using Isolated Storage) when the analytics data endpoint is unavailable. If the value of this property is **None** then application usage data will not be stored when the analytics data endpoint is unavailable.
OfflineStateSourceName : String	The name of the property, field, or method parameter that will contain the offline state of the application at runtime, at the time that the setup method is called. If this property is set, `OfflineStateSourceElement` is required to be set as well.
OfflineStateSourceOwner : Type	If the `OfflineStateSourceElement` is a field or property, `OfflineStateSourceOwner` indicates the name of the type that defines the field or property. If not set, the named source element is searched for on the attributed method's type.
OfflineStorageResultSinkElement : SourceElements	Indicates how the application is notified when usage data is stored to the offline storage mechanism. If this property is not DefaultAction or None then `OfflineStorageResultSinkName` must also be set. If `OfflineStorageResultSinkElement` is a field or property, `OfflineStorageResultSinkOwner` must also be set unless the field or property is defined on the same class as the attributed method. If the value of this property is **None** then no notification will take place when application usage data is saved to the offline storage mechanism. If the value of this property is

	DefaultAction then the application will exit immediately when a storage failure occurs
OfflineStorageResultSinkName : String	The name of the property, field, or method parameter that will contain the most recent result of the application storing usage data to the offline storage mechanism If this property is set, `OfflineStorageResultSinkElement` is required to be set as well.
OfflineStorageResultSinkOwner : Type	If the `OfflineStorageResultSinkElement` is a field or property, `OfflineStorageResultSinkOwner` indicates the name of the type that defines the field or property. If not set, the named source element is searched for on the attributed method's type.
OptInSourceElement : SourceElements	Indicates how to access the opt-in flag at runtime, at the time that the setup method is called (e.g. a field, property, or method parameter). To use this property, `OptInSourceName` must also be set. If `OptInSourceElement` is a field or property, `OptInSourceOwner` must also be set unless the field or property is defined on the same class as the attributed method.
OptInSourceName : String	The name of the property, field, or method parameter that will contain the opt-in flag at runtime, at the time that the setup method is called. If this property is set, `OptInSourceElement` is required to be set as well. If this property is not set, Dotfuscator will generate code that sends PreEmptive Analytics messages all the time.
OptInSourceOwner : Type	If the `OptInSourceElement` is a field or property, `OptInSourceOwner` indicates the name of the type that defines the field or property. If not set, the named source element is searched for on the attributed method's type.
StaticEndpoint : String	Defines the destination URI for PreEmptive Analytics messages. This URI must represent the endpoint for a web service which will consume PreEmptive Analytics messages. If this property is not set, null, or empty, PreEmptive Analytics messages will be sent to the well-known URI representing PreEmptive's commercial Runtime Intelligence Services endpoint.
UseSSL : Boolean	Use HTTPS protocol when sending PreEmptive Analytics messages to a PreEmptive Analytics Endpoint. Default value is true.

2.6.4.1.11 SinkElements

Summary

Possible sinks for data meant to be provided by instrumented code. Examples include the application tamper notification specified in the InsertTamperCheckAttribute.

SinkElements

```
public enum SinkElements : Attribute
```

Enumeration Members

Field	Summary
Field	The sink element is a field.
MethodArgument	The sink element is a method argument that is a Delegate type.
Method	The sink element is a method to be called.
None	No sink element. Individual attributes might implement the default action’s behavior if no sink element is specified. See the attribute documentation for details.
Property	The sink element is a settable property.
Delegate	The sink element is a field that is a field that is a Delegate type.
DefaultAction	Perform the default action when a sink is required. Individual attributes have different default actions.

2.6.4.1.12 SourceElements

Summary

Possible sources for data meant to be consumed by PreEmptive Analytics generated code. Examples include the application instance ID and opt-in flag specified in the SetupAttribute.

SourceElements

```
public enum SourceElements : Attribute
```

Enumeration Members

Field	Summary
Field	The source element is a field.
Method	The source element is a method to be called.
MethodArgument	The source element is a method argument.
None	No source element. This is the default value.
Property	The source element is a property.

2.6.4.1.13 SystemProfileAttribute

Summary

The SystemProfileAttribute is an instrumentation time attribute for methods. Dotfuscator will insert code to generate and send a PreEmptive Analytics Profile Message into any method with this attribute. Dotfuscator will remove this custom attribute from the metadata after instrumentation. This attribute has no properties outside the optional **`ExtendedKeysSource`** properties that all PreEmptive Analytics attributes have.

SystemProfile Attribute

```
public class SystemProfileAttribute : Attribute
```

Constructor Members

Name	Summary
SystemProfileAttribute()	Initializes a new instance of the class.

Property Members

Name	Summary
ExtendedKeyMethodArguements : string	A pattern indicating which parameter names and values should be automatically added to the messages extended key data at runtime. See Automatically Sending Method Parameters as Extended Keys for details on supported patterns.
ExtendedKeySourceElement : SourceElements	Indicates how to access the extended key dictionary at runtime, at the time that the attributed method is called (e.g. a field, property, method, or method parameter). To use this property, `ExtendedKeySourceName` must also be set. If `ExtendedKeySourceElement` is a field, method, or property, `ExtendedKeySourceOwner` must also be set unless the field, method, or property is defined on the same class as the attributed method.
ExtendedKeySourceName : string	The name of the property, field, method, or method parameter that will contain the extended key dictionary at runtime, at the time that the attributed method is called. If this property is set, `ExtendedKeySourceElement` is required to be set as well. If this property is not set, Dotfuscator will not generate code to send extended key information.
ExtendedKeySourceOwner : Type	If the `ExtendedKeySourceElement` is a field, method or property, `ExtendedKeySourceOwner` indicates the name of the type that defines the field, method or property. If not set, the named source element is searched for on the attributed method's type.

2.6.4.1.14 TeardownAttribute

Summary

The TeardownAttribute is an instrumentation time attribute for methods. This attribute can be used to guide Application Analytics instrumentation.

When used for Application Analytics, Dotfuscator will insert PreEmptive Analytics cleanup code into any method with this attribute. There must be one or more methods with this attribute in an assembly or application that uses PreEmptive Analytics instrumentation. At runtime, the cleanup code will send a PreEmptive Analytics application and session stop messages when this method is called.

Dotfuscator will remove this custom attribute from the metadata after instrumentation.

TeardownAttribute

```
public class TeardownAttribute : Attribute
```

Constructor Members

Name	Summary
TeardownAttribute()	Initializes a new instance of the class.

Property Members

Name	Summary
ExtendedKeyMethodArguements : string	A pattern indicating which parameter names and values should be automatically added to the messages extended key data at runtime. See Automatically Sending Method Parameters as Extended Keys for details on supported patterns.
ExtendedKeySourceElement : SourceElements	Indicates how to access the extended key dictionary at runtime, at the time that the attributed method is called (e.g. a field, property, method, or method parameter). To use this property, `ExtendedKeySourceName` must also be set. If `ExtendedKeySourceElement` is a field, method, or property, `ExtendedKeySourceOwner` must also be set unless the field, method or property is defined on the same class as the attributed method.
ExtendedKeySourceName : string	The name of the property, field, method, or method parameter that will contain the extended key dictionary at runtime, at the time that the attributed method is called. If this property is set, `ExtendedKeySourceElement` is required to be set as well. If this property is not set, Dotfuscator will not generate code to send extended key information.
ExtendedKeySourceOwner : Type	If the `ExtendedKeySourceElement` is a field, method or property, `ExtendedKeySourceOwner` indicates the name of the type that defines the field, method or property. If not set, the named source element is searched for on the attributed method’s type.

2.6.5 The Map File

Dotfuscator generates a mapping file that associates old with new names. The new names of the classes, methods, and fields are shown. Bug tracking becomes difficult after renaming, especially with a high incidence of method overloading, making the map file essential.

The map file can be used to Decode Obfuscated Stack Traces as well as for Incremental Obfuscation. The map file also provides statistics regarding the overall effectiveness of renaming.

The elements of the mapping file are all very similar. A few things are noteworthy:

- If a `<newname>` element is absent, then the item was not renamed.
- In type names, nested class names are separated from the parent using the “`/`” character.
- Constructors are named `.ctor`, while static constructors (a.k.a. static initializers, class constructors, etc) are named `.cctor`. These are never renamed.

For additional reference, see the governing DTD for the map file.

2.6.5.1 dotfuscatorMap_v1.1.dtd

The dotfuscatorMap_v1.1.dtd DTD describes the format of the renaming map file produced and consumed by Dotfuscator version. A copy is available at www.preemptive.com/dotfuscator/dtd/dotfuscatorMap_v1.1.dtd.

2.6.6 Advanced Topics

The following sections detail advanced Dotfuscator usage.

2.6.6.1 Side by Side Installs

Multiple versions of Dotfuscator can be installed side by side from version 4.11 and up. To enable this click on the "advanced" button in the installer wizard and select "Install Side by Side With Current Installations". You can also configure which version of Dotfuscator will be used by Visual Studio by default for Dotfuscator Projects. You can select "latest Major", "latest Major.Minor", or the specific version you are installing. You can change the default later by editing the registry key at HKCU\Software\PreEmptive\Dotfuscator\DefaultMSBuildPath.

MSBuild Extensions

If you uninstall the most recent version, it will leave behind its MSBuild task and target files in the major and major.minor directories. To revert the major or major.minor files to a prior version you should either copy them from a specific version's subdirectory or manually delete the files and perform a repair on the version you want to revert to through Add/Remove Programs.

Visual Studio Integration

When doing a side by side installation, you can disable the Visual Studio integration features to not overwrite the current Visual Studio integrated Dotfuscator. If the version of Dotfuscator that is integrated with Visual Studio is uninstalled and you wish to revert to another version for integration, go into Add/Remove programs and select change on that version of Dotfuscator and first disable the integration and complete and then rerun the change with the feature enabled.

Please note that the procedure given here will allow multiple versions of Dotfuscator to be run via the command line, via MSBuild, and as the standalone Dotfuscator UI, but only one version of Dotfuscator may be integrated with Visual Studio at a time.Because a Dotfuscator Visual Studio Project is compatible with MSBuild, you can use a single .dotfuproj file for running in both contexts. However, Visual Studio integration is not capable of using the version of Dotfuscator pointed to by the DotfuscatorBinPath property. Visual Studio integration will always build using the latest installed version of Dotfuscator. If the DotfuscatorBinPath is not the default, you will get the following warning: "This Dotfuscator Project has a DotfuscatorBinPath or DotfuscatorDataPath different from the installed Visual Studio Dotfuscator extension."

2.6.6.2 Concurrent Builds

When running multiple copies of Dotfuscator simultaneously, it is recommended that the following files be copied into the same directory as the Dotfuscator executable:

```
dotfuscator.dat (normally in %PROGRAMDATA%\PreEmptive Solutions\Dotfuscator
[Edition]\4.0)
dfusrprf.xml    (normally in %LOCALAPPDATA%\PreEmptive Solutions\Dotfuscator
[Edition]\4.0)
dotfuscator.cfg (normally in %LOCALAPPDATA%\PreEmptive Solutions\Dotfuscator
[Edition]\4.0)
```

2.7 Samples

This section contains descriptions of additional sample applications that are designed to show you how to configure Dotfuscator for various types of .NET applications.

If you are looking for a simple example that will let you quickly get familiar with Dotfuscator, see Getting Started with Dotfuscator.

The samples here are explained using C#. Samples explained using VB.NET are available on our web site and in the Dotfuscator installation directory. If your installer did not place these samples in your Dotfuscator installation directory, the sample files are also freely available for download at www.preemptive.com/dotfuscator-samples.html.

2.7.1 Reflection Sample

The reflection sample demonstrates issues that occur when using Dotfuscator with applications that make use of dynamic class loading and method invocation. These powerful technologies allow applications to delay specification of code to be executed until runtime. In these cases, it is impossible to statically predict what classes and methods might be invoked at runtime; therefore, it is impossible for Dotfuscator to infallibly determine which identifiers should be excluded from renaming. Fortunately, Dotfuscator has rich facilities for the fine-tuning of renaming rules.

The reflection sample demonstrates how to configure Dotfuscator to selectively exclude types and methods from renaming. Instructions are included for using both the command line and the graphical interface of Dotfuscator.

2.7.1.1 Reflection Sample Files

The reflection sample includes the following files:

File	Description
Reflection.doc	This document
Reflection.cs	C# file which demonstrates reflection techniques
reflection_config.xml	Dotfuscator configuration file
make.bat	Batch file for compiling the reflection application
run.bat	Batch file to run Dotfuscator on the application

The reflection sample files can be downloaded at www.preemptive.com/dotfuscator-samples.html.

2.7.1.2 Building the Reflection Sample

The reflection sample assumes that the C# compiler (**csc.exe**) is reachable from your PATH environment variable. If you are using Visual Studio, you can make sure that it is in your path by executing the **Visual Studio Command Prompt** shortcut in your start menu.

From the command prompt, change your current directory to the directory containing the reflection sample.

Build the application by executing **make.bat**. This batch file will invoke the Visual C# compiler to produce the output file - **Reflection.exe**.

2.7.1.3 Running the Reflection Sample

The reflection sample can be run by executing the Reflection.exe assembly produced by the **make.bat** command. The reflection program dynamically loads a class from the current assembly with the following code:

Reflection Sample Code

```
//get the requested type from current assembly
 assembly = this.GetType().Assembly;
 type = assembly.GetType(typename, true);
 instance = Activator.CreateInstance(type);
```

It is worth noting that the class that is loaded is specified by the string variable typename. Looking closer at the sample code shows us that `typename` is initialized to the value `Samples.Greeting`.

Further on in the program, the `SayGreetings` method of `Samples.Greeting` is invoked dynamically with similar code:

Sample.Greeting Code

```
   MethodInfo method = type.GetMethod(methodname);
   …
   method.Invoke(instance,methodargs);
```

As its name suggests, the **`SayGreetings`** method displays some friendly messages on the console:

SayGreetings Friendly Messages

```
Hello Bob!
Goodbye Bob!
```

2.7.1.4 Dotfuscating the Reflection Output

The reflection sample contains a sample Dotfuscator configuration file that demonstrates using exclusion rules to exclude these items invoked by reflection. This file is named **reflection_config.xml** and can be located in the same directory as the rest of the reflection samples. The section of the file that excludes these references is:

Using Exclusion Rules to Exclude Items Invoked by Reflection

```
<renaming>
   <excludelist>
      <type name="Samples.Greetings">
         <method name="SayGreetings" signature="string" />
      </type>
   </excludelist>
…
</renaming>
```

The **`<renaming>`** tag indicates that the exclusion rules contained within pertain specifically to identifier renaming, as opposed to other Dotfuscator features which can also be selectively turned on or off.

The **`<excludelist>`** tag defines a list of items that must be excluded from the renaming process. The **`<type name=`**"**`Samples.Greetings`**"**`>`** tag instructs Dotfuscator to exclude the class name "**`Samples.Greetings`**" from the renaming process. Note that this only refers to the class name itself. All methods and fields belonging to the "**`Greetings`**" class are still eligible to be renamed unless they are specifically excluded. We can see an example of this with the **`<method name="SayGreetings" signature="string" />`** entry.

Executing the **make.bat** file will run Dotfuscator with this configuration file. The output of this process is a Reflection.exe assembly in the "**`output`**" subdirectory. This location can be controlled by modifying the following section in the configuration file:

Modifying Location of the Output

```
<output>
 <file dir="${projectdir}\output" />
</output>
```

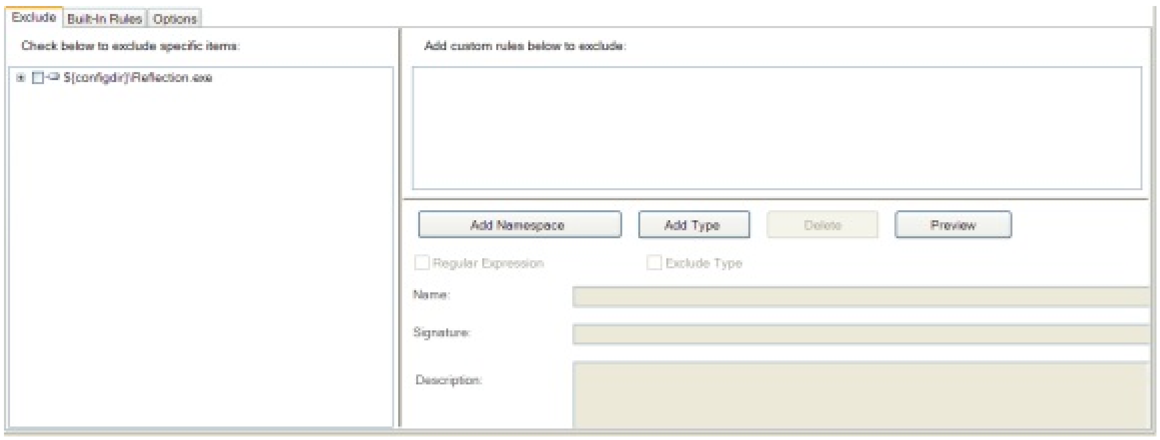
Running the new assembly verifies that Dotfuscator correctly excluded the required items from the renaming process:

Friendly Messages Excluded from Renaming Process:

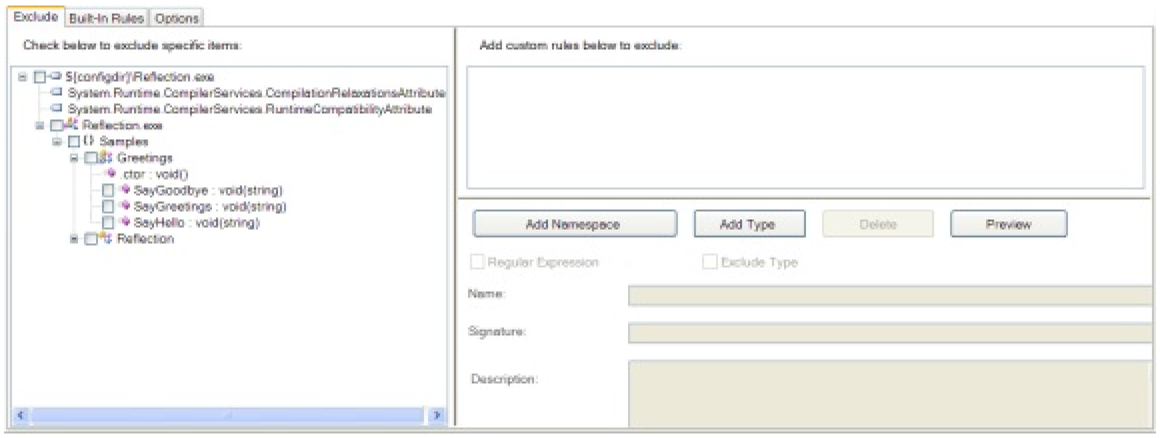
```
Hello Bob!
Goodbye Bob!
```

2.7.1.5 Configuring the Reflection Sample with the Graphical User Interface

The Dotfuscator graphical interface provides a visual means to produce a configuration file. Items to be excluded from renaming can be specified using the *Rename* tab of the interface:



Expanding the assembly node in the tree shows a graphical view of the application structure, including all namespaces, types, and methods:

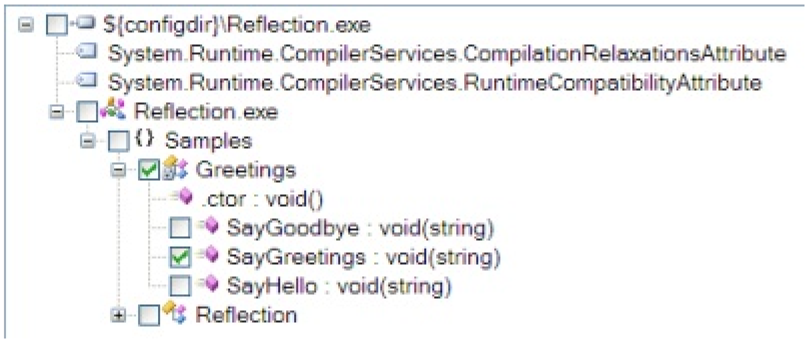


Check below to exclude specific items:

```
${configdir}\Reflection.exe
    System.Runtime.CompilerServices.CompilationRelaxationsAttribute
    System.Runtime.CompilerServices.RuntimeCompatibilityAttribute
  Reflection.exe
    Samples
      Greetings
        .ctor : void()
        SayGoodbye : void(string)
        SayGreetings : void(string)
        SayHello : void(string)
      Reflection
```

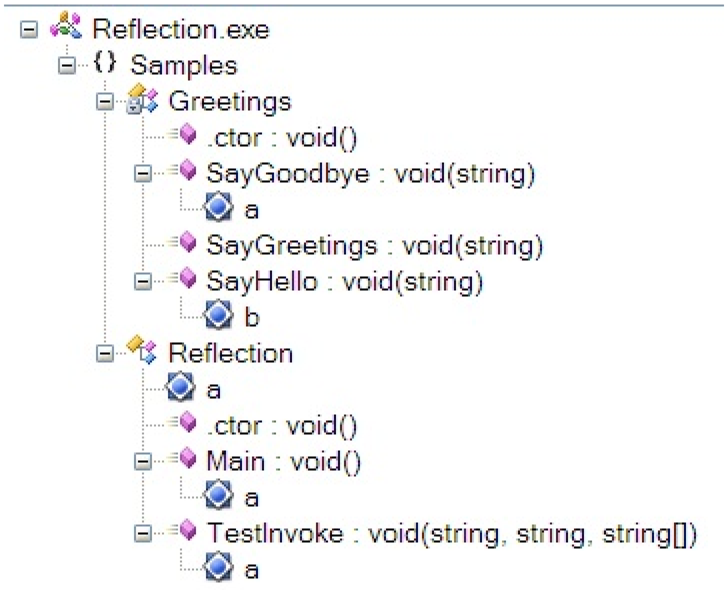
Add custom rules below to exclude:

Graphically generating a renaming exclusion list is a simple matter of checking the boxes next to the item to be excluded. In our reflection example, the required exclusions are made by checking the box next to the type "**`Greetings`**" and the method "**`SayGreetings`**":



```
[ ] ${configdir}\Reflection.exe
      System.Runtime.CompilerServices.CompilationRelaxationsAttribute
      System.Runtime.CompilerServices.RuntimeCompatibilityAttribute
  [ ] Reflection.exe
    [ ] Samples
      [✓] Greetings
            .ctor : void()
          [ ] SayGoodbye : void(string)
          [✓] SayGreetings : void(string)
          [ ] SayHello : void(string)
      [ ] Reflection
```

Building the project produces the correct output that can be verified with the *Output* tab of the application:



```
Reflection.exe
  Samples
    Greetings
      .ctor : void()
      SayGoodbye : void(string)
        a
      SayGreetings : void(string)
      SayHello : void(string)
        b
    Reflection
      a
      .ctor : void()
      Main : void()
        a
      TestInvoke : void(string, string, string[])
        a
```

2.7.1.6 Summary of the Reflection Sample

In order for you to successfully Dotfuscate an application that loads classes by name, invokes methods by name, or references fields by name, you need to manually exclude the appropriate identifiers from renaming. Dotfuscator provides a fine-grained, rule based facility for doing this.

2.7.2 Serialization Sample

The serialization sample demonstrates using Dotfuscator in an application that makes use of serialized objects that must be exchanged with non-obfuscated code. If your obfuscated application is the only producer or consumer of the serialized objects, then this sample does not apply.

Serialization must be considered when using Dotfuscator because all of the provided serialization formatters embed type and field information in the persisted data stream. The implication of this is that classes and fields that are serialized must be excluded from renaming, otherwise it would be impossible to de-serialize objects persisted with non-obfuscated code. Since method definitions are not persisted, they do not need to be excluded from renaming.

The serialization sample demonstrates how to configure Dotfuscator to selectively exclude types and fields from renaming. Instructions are included for using both the command line and the graphical interface.

2.7.2.1 Serialization Sample Files

The serialization sample includes the following files:

File	Description
Serialization.doc	This document
Serialization.cs	C# file which demonstrates serialization techniques
serialization_config.xml	Dotfuscator configuration file
make.bat	Batch file for compiling the serialization application
run.bat	Batch file to run Dotfuscator on the application

The serialization sample files can be downloaded at www.preemptive.com/dotfuscator-samples.html.

2.7.2.2 Building the Serialization Sample

The serialization sample assumes that the C# compiler (**csc.exe**) is reachable from your PATH environment variable. If you are using Visual Studio, you can make sure that it is in your path by executing the **Visual Studio Command Prompt** shortcut in your start menu.

From the command prompt, change your current directory to the directory containing the serialization sample.

Build the application by executing **make.bat**. This batch file will invoke the Visual C# compiler to produce the output file - **Serialization.exe**.

2.7.2.3 Running the Serialization Sample

The serialization sample can be run by executing the Serialization.exe assembly produced by the make.bat command. The application has two modes of operation controlled by command line parameters. The "**-s**" option, causes the application to instantiate an object of type "**Tester**", and persist it to the file "**Sum.out**". Each Tester object has 3 integer fields that make up the persisted data. The following code from the Tester class is used to persist the object:

Tester Class Code

```
// create a file stream to write the file
FileStream stream = new fileStream("Sum.out",FileMode.Create);

BinaryFormatter formatter = new BinaryFormatter();
// serialize to disk
formatter.Serialize(stream,this);
stream.Close();
```

The "**-d**" option, causes the application to de-serialize an object that had previously been persisted in the file "**Sum.out**". The following code is responsible for doing the de-serialization:

De-Serialization Code

```
stream = new FileStream("Sum.out",FileMode.Open);
BinaryFormatter formatter = new BinaryFormatter();
return (Tester) formatter.Deserialize(stream);
```

2.7.2.4 Dotfuscating the Serialization Output

The serialization sample contains a sample Dotfuscator configuration file that demonstrates using exclusion rules to exclude the information that makes up the persisted representation of an object. This file is named **serialization_config.xml** and can be located in the same directory as the rest of the serialization samples. The section of the file that excludes these references is:

Serializaiton Sample Reference File

```
<renaming>
<excludelist>
  <type name="Samples.Tester">
   <field name=".*" regex="true" />
  </type>
</excludelist>
...
</renaming>
```

The **<renaming>** tag indicates that the exclusion rules contained within pertain specifically to identifier renaming, as opposed to other Dotfuscator features which can also be selectively turned on or off.

The **<excludelist>** tag defines a list of items that must be excluded from the renaming process. The **<type name=**"**Samples.Tester**"**>** tag instructs Dotfuscator to exclude the class name "**Samples.Tester**" from the renaming process. Note that this only refers to the class name itself. All methods of the "**Tester**" class are still eligible for renaming. The **<field name=".*" regex="true" />** tag instructs Dotfuscator to exclude all fields contained in the Tester class. Instead of calling out each field individually, which would become unwieldy in a large class, this example uses a regular expression to specify exclusion. In this case, it uses the very simple expression "**.***" which matches all fields.

Executing the make.bat file will run Dotfuscator with this configuration file. The output of this process is a **Serialization.exe** assembly in the "output" subdirectory. This location can be controlled by modifying the following section in the configuration file:

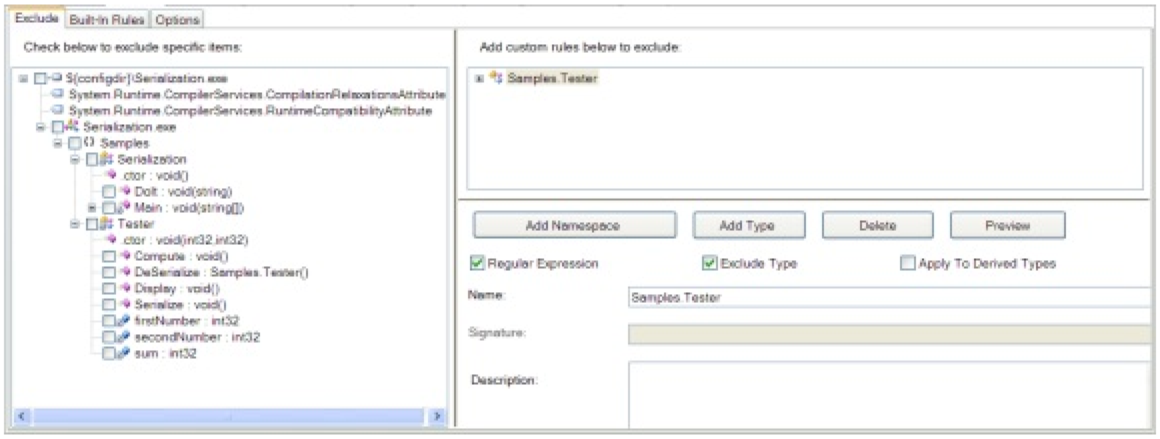
Example Title

```
<output>
<file dir="${projectdir}\output" />
</output>
```

Running the new assembly verifies that Dotfuscator correctly excluded the required items from the renaming process. It is also important to note that the obfuscated application can successfully de-serialize objects persisted with the non-obfuscated application, and the non-obfuscated application can de-serialize objects persisted with the obfuscated application. The serialized objects are completely compatible with one another.

2.7.2.5 Configuring the Serialization Sample with the Graphical User Interface

The Dotfuscator graphical interface provides a visual means to produce a configuration file. Items to be excluded from renaming can be specified using the *Rename* tab of the interface. Expanding the assembly node in the tree shows a graphical view of the application structure, including all namespaces, types, and methods:



Check below to exclude specific items:

```
$(configdir)\Serialization.exe
  System.Runtime.CompilerServices.CompilationRelaxationsAttribute
  System.Runtime.CompilerServices.RuntimeCompatibilityAttribute
  Serialization.exe
    Samples
      Serialization
        .ctor : void()
        DoIt : void(string)
        Main : void(string[])
      Tester
        .ctor : void(int32,int32)
        Compute : void()
        DeSerialize : Samples.Tester()
        Display : void()
        Serialize : void()
        firstNumber : int32
        secondNumber : int32
        sum : int32
```

Add custom rules below to exclude:

Samples.Tester

[✓] Regular Expression [✓] Exclude Type [ ] Apply To Derived Types

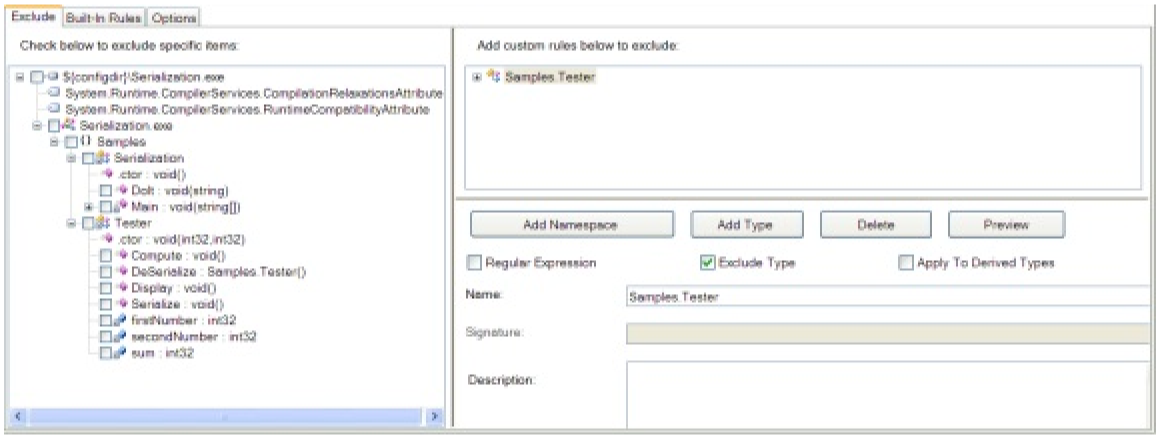
Name: Samples.Tester

Signature:

Description:

Graphically generating a renaming exclusion list is a two-step process in order to use the regular expression capability of the interface. First, create a type exclusion rule to match on the type "`Tester`". Do this by pressing the **Add Type** button. This adds a new node to the right-hand panel named simply **Type**.

Next, change the name field of this new node to **Samples.Tester**, reflecting the class to be excluded. Also, deselect the regular expression option since there is only one class to be excluded:



Check below to exclude specific items:

```
$(configdir)\Serialization.exe
  System.Runtime.CompilerServices.CompilationRelaxationsAttribute
  System.Runtime.CompilerServices.RuntimeCompatibilityAttribute
  Serialization.exe
    Samples
      Serialization
        .ctor : void()
        DoIt : void(string)
        Main : void(string[])
      Tester
        .ctor : void(int32,int32)
        Compute : void()
        DeSerialize : Samples.Tester()
        Display : void()
        Serialize : void()
        firstNumber : int32
        secondNumber : int32
        sum : int32
```

Add custom rules below to exclude:

Samples.Tester

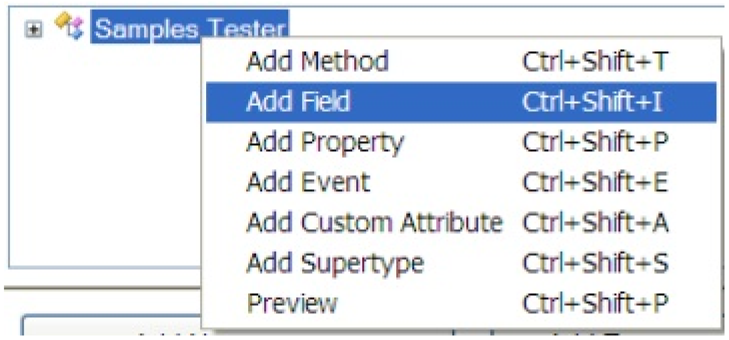
[ ] Regular Expression [✓] Exclude Type [ ] Apply To Derived Types

Name: Samples.Tester

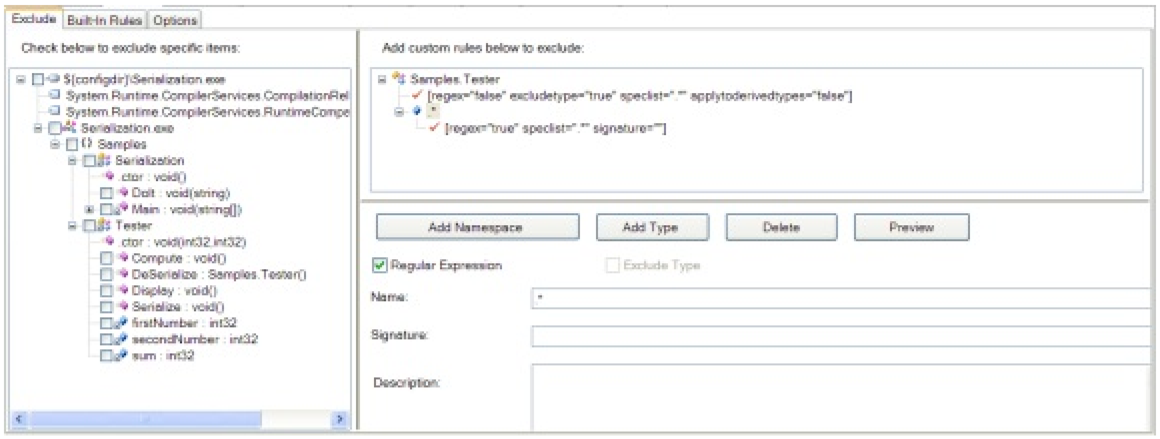
Signature:

Description:

To add the field exclusion rule, right click on the newly added **Samples.Tester** node. This will bring up a context menu.



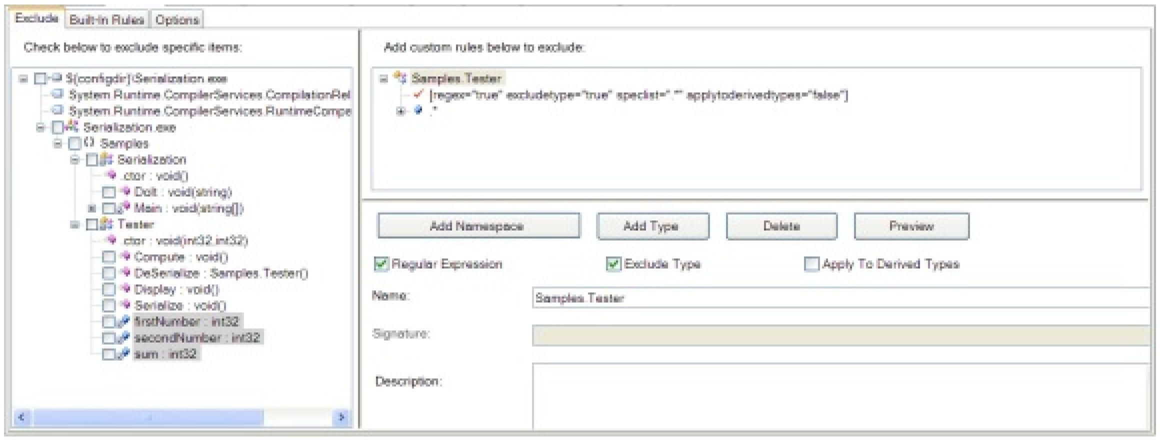
Choosing **Add Field**, causes a new node to be added to the tree. Change the name to "**.***" to indicate that this expression should match all fields:



```
Samples.Tester
  [regex="false" excludetype="true" speclist="" applytoderivedtypes="false"]
  *
    [regex="true" speclist="*" signature=""]
```

[✓ Regular Expression] [ ] Exclude Type
Name: *
Signature:

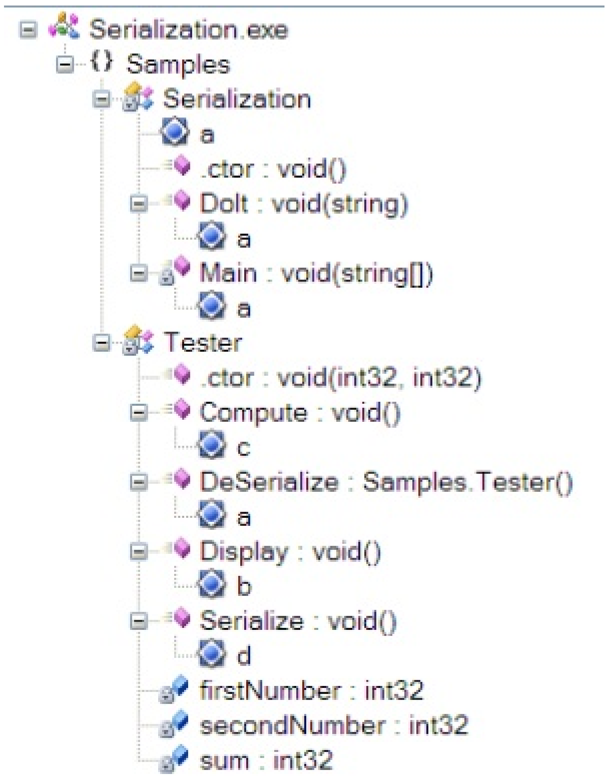
Pressing the **Preview** button applies the rules and shows the items that will be excluded in gray in the left-hand tree. A quick inspection shows this to be the desired outcome, with all fields from **Samples.Tester** excluded:



```
Samples.Tester
  [regex="true" excludetype="true" speclist="" applytoderivedtypes="false"]
  *
```

[✓ Regular Expression] [✓ Exclude Type] [ ] Apply To Derived Types
Name: Samples.Tester

Building the project produces the correct output that can be verified with the output tab of the application:



```
Serialization.exe
  Samples
    Serialization
      a
      .ctor : void()
      DoIt : void(string)
        a
      Main : void(string[])
        a
    Tester
      .ctor : void(int32, int32)
      Compute : void()
        c
      DeSerialize : Samples.Tester()
        a
      Display : void()
        b
      Serialize : void()
        d
      firstNumber : int32
      secondNumber : int32
      sum : int32
```

2.7.2.6 Summary of the Serialization Sample

In order to Dotfuscate an application that exchanges serialized objects with external applications, you need to make sure that the appropriate data elements are excluded from the renaming process. Dotfuscator provides extremely powerful mechanism for defining these exclusion rules. By following these guidelines, you can help ensure that your Dotfuscated application can safely exchange data with non-Dotfuscated applications with ease.

2.7.3 Remoting Sample

The remoting sample demonstrates using Dotfuscator in an application that makes use of .NET Remoting. In a typical remoting application, objects that are to service remote invocations are registered with the runtime before they can be invoked. When an invocation request arrives, the runtime instantiates a server object dynamically using reflection. Since this process happens at runtime, there is no way for Dotfuscator to know at analysis time what types must be excluded. Manual user intervention is required to make sure that Dotfuscator does not rename remoted types.

The remoting sample demonstrates how to configure Dotfuscator to selectively exclude types from renaming. This sample also demonstrates how to configure multiple assemblies in the same Dotfuscator project. Instructions are included for using both the command line and the graphical interface of Dotfuscator.

2.7.3.1 Remoting Sample Files

The remoting sample includes the following files:

Files	Description
Remoting.doc	This document
TrigServer.cs	File containing code for the remoting server
TrigClient.cs	File containing code for the remoting client
ITrigFunctions.cs	File containing the common interface
remoting_config.xml	Dotfuscator configuration file
make.bat	Batch file for compiling the remoting application
run.bat	Batch file to run Dotfuscator on the application

The remoting sample files can be downloaded at www.preemptive.com/dotfuscator-samples.html.

2.7.3.2 Building the Remoting Sample

The remoting sample assumes that the C# compiler (**csc.exe**) is reachable from your PATH environment variable. If you are using Visual Studio, you can make sure that it is in your path by executing the **Visual Studio Command Prompt** shortcut in your start menu.

From the command prompt, change your current directory to the directory containing the remoting sample.

Build the application by executing make.bat. This batch file will invoke the Visual C# compiler to produce the output files – **TrigServer.exe**, **TrigClient.exe**, **ITrigFunctions.dll**.

2.7.3.3 Running the Remoting Sample

The first step to running the remoting sample is to launch the **TrigServer.exe** assembly. This application will register a new HTTP channel listener on port 12345, and register the class `Samples.Trig` to handle method invocations on that channel. The code from TrigServer that does this is:

TrigServer Code

```
   // create a channel and start listening
   HttpChannel serverchannel = new HttpChannel(12345);
   ChannelServices.RegisterChannel(serverchannel);

   Type trigType = Type.GetType("Samples.Trig");

   // register our well-known type
   RemotingConfiguration.RegisterWellKnownServiceType(
      trigType,
      "TrigServer",
      WellKnownObjectMode.Singleton );
```

Once the server is established, the client application, **TrigClient.exe**, can be run. This application establishes an outgoing HTTP channel and connects it to the TrigServer as can be seen in the following code:

Connect to TrigServer Code

```
   // create a channel for our client call
   HttpChannel clientchannel = new HttpChannel(0);
   ChannelServices.RegisterChannel(clientchannel);

   // get a reference to the remote Trig server
   MarshalByRefObject rawobject =
      (MarshalByRefObject)RemotingServices.Connect(
         typeof(Samples.ITrigFunctions),
         "http://localhost:12345/TrigServer");
```

After establishing the connection, the client invokes several methods on the remote server object:

Invoked Methods on Remote Server Object

```
   Console.WriteLine("Cos(0) = {0}", trig.Cos(0.0));
   Console.WriteLine("Sin(0) = {0}", trig.Sin(0.0));
   Console.WriteLine("Cos(PI) = {0}", trig.Cos(trig.PI()));
```

The output of the client application demonstrates correct operation of the application:

Output

```
About to make connection to remote server
Cos(0) = 1
Sin(0) = 0
Cos(PI) = -1
```

2.7.3.4 Dotfuscating the Remoting Output

The remoting sample contains a sample Dotfuscator configuration file that demonstrates using multiple input assemblies and using exclusion rules to exclude the type information for the object invoked through remoting. The multiple input assemblies are indicated in the configuration file with multiple **`<file>`** entries in the **`<input>`** assemblies list as shown:

Using Multiple Input Assemblies and Exclusion Rules

```
<input>
  <filelist>
   <file dir="${projectdir}" name="ITrigFunctions.dll"/>
   <file dir="${projectdir}" name="TrigServer.exe"/>
   <file dir="${projectdir}" name="TrigClient.exe"/>
  </filelist>
</input>
```

The other interesting aspect of the configuration file is the renaming exclusion list:

Renaming Exclusion List

```
<renaming>
  <excludelist>
   <type name="Samples.Trig" excludetype="true"/>
  </excludelist>
…
</renaming>
```

The **`<renaming>`** tag indicates that the exclusion rules contained within pertain specifically to identifier renaming, as opposed to other Dotfuscator features which can also be selectively turned on or off.

The **`<excludelist>`** tag defines a list of items that must be excluded from the renaming process. The **`<type name=`**"**`Samples.Trig`**"**`>`** tag instructs Dotfuscator to exclude the class name "**`Samples.Trig`**" from the renaming process. Note that this only refers to the class name itself. All methods of the “Trig” class are still eligible for renaming. This is acceptable since we are Dotfuscating both the client and the server in this case. If we were creating an interface that was going to be called by clients we did not Dotfuscate, we would also want to prevent renaming of the interface methods as well. Since we included all assemblies in the same project, Dotfuscator will be able to rename method call references to the appropriate renamed server method call.

Executing the **make.bat** file will run Dotfuscator with this configuration file. The outputs of this process are **TrigServer.exe**, **TrigClient.exe** and **ITrigFunctions.dll** assemblies in the "**`output`**" subdirectory. This location can be controlled by modifying the following section in the configuration file:

TrigServer

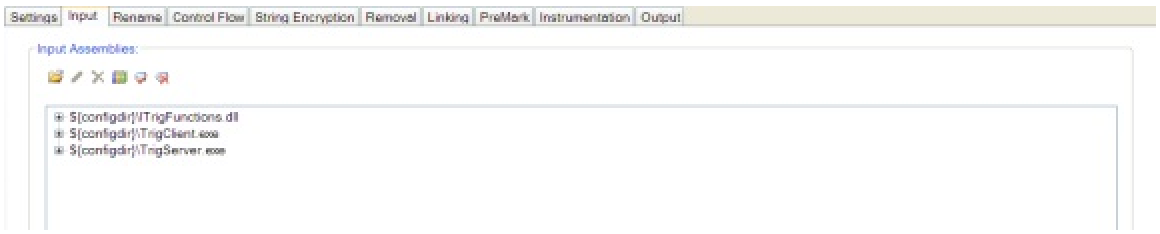
```
<output>
  <file dir="${projectdir}\output" />
</output>
```

Note: the 1:1 relationship between input and output assemblies cannot be changed.

Running the new assemblies verifies that Dotfuscator correctly excluded the required items from the renaming process. It is also important to note that since we allowed the renaming of interface methods, it is not possible to have an obfuscated client call a non-obfuscated server or vice-versa.

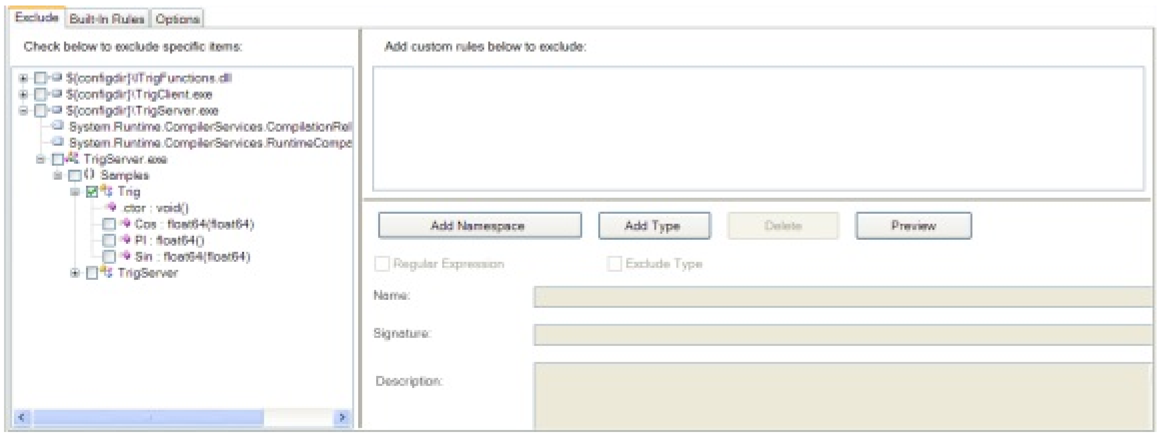
2.7.3.5 Configuring the Remoting Sample with the Graphical User Interface

The remoting sample makes use of Dotfuscator's ability to have multiple input assemblies. Multiple input assemblies can be added with the graphical interface easily. Simply use the browse facility to locate each assembly individually. All input assemblies will appear in the *Input Assemblies:* list box:



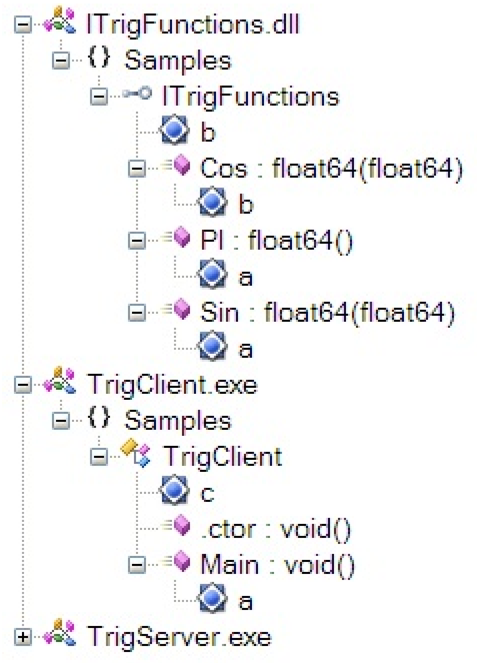
```
⊞ $(configdir)\\TrigFunctions.dll
⊞ $(configdir)\TrigClient.exe
⊞ $(configdir)\TrigServer.exe
```

The type "**`Samples.Trig`**" which is defined in the **`TrigServer`** assembly must be excluded from renaming since it is registered via reflection. The following figure shows this item selected for exclusion in the renaming tab of the interface:



```
⊞ ☐ $(configdir)\\TrigFunctions.dll
⊞ ☐ $(configdir)\TrigClient.exe
⊟ ☐ $(configdir)\TrigServer.exe
      System.Runtime.CompilerServices.CompilationRel
      System.Runtime.CompilerServices.RuntimeCompa
   ⊟ ☐ TrigServer.exe
      ⊟ ☐ {} Samples
         ⊟ ✓ Trig
               .ctor : void()
            ☐ Cos : float64(float64)
            ☐ PI : float64()
            ☐ Sin : float64(float64)
         ⊞ ☐ TrigServer
```

After building, the output tab shows the results. Notice that everything except for the **`Samples.Trig`** class was renamed.



```
ITrigFunctions.dll
  Samples
    ITrigFunctions
      b
      Cos : float64(float64)
        b
      PI : float64()
        a
      Sin : float64(float64)
        a
TrigClient.exe
  Samples
    TrigClient
      c
      .ctor : void()
      Main : void()
        a
TrigServer.exe
```

2.7.3.6 Summary of the Remoting Sample

In order to Dotfuscate an application which serves objects to clients through remoting, it is important to exclude the type names of the served objects from the renaming process. The power of Dotfuscator is enhanced in a remoting situation if both the client and server are Dotfuscated together. In these cases, Dotfuscator can be aggressive and rename the external interfaces of the components in addition to the internal methods and fields.

2.7.4 ASP.NET Sample

ASP.NET introduces the concept of code-behind to the suite of tools available to web developers. Code-behind enables the development of a web application’s logic in any .NET language. This logic can then be pre-compiled into .NET assemblies that are loaded on demand by the web server. Dotfuscator is able to process these assemblies and give your web applications an added layer of protection above what is already provided by your server infrastructure.

The ASP.NET sample demonstrates how to configure Dotfuscator to selectively exclude types and fields from renaming. Instructions are included for using both the command line and the graphical interface of Dotfuscator.

2.7.4.1 ASP.NET Sample Files

The ASP.NET sample includes the following files:

File	Description

asp.doc	This document
DotfuscatorASP.aspx	Main sample page
DotfuscatorASP.aspx.cs	C# code-behind file containing code to be Dotfuscated
asp_config.xml	Dotfuscator configuration file
make.bat	Batch file for compiling the serialization application
run.bat	Batch file to run Dotfuscator on the application
deploy.bat	Batch file to copy the obfuscated assemblies to the web-accessible location

The asp.net sample files can be downloaded at www.preemptive.com/dotfuscator-samples.html.

2.7.4.2 Preparing the ASP.NET Sample

This sample requires that IIS version 5.1 or greater be installed and configured on the target machine. It is necessary to create a virtual directory in IIS configuration that references the location where the asp.net samples are installed. Refer to the IIS documentation for instructions on how to set up a virtual directory.

2.7.4.3 Building the ASP.NET Sample

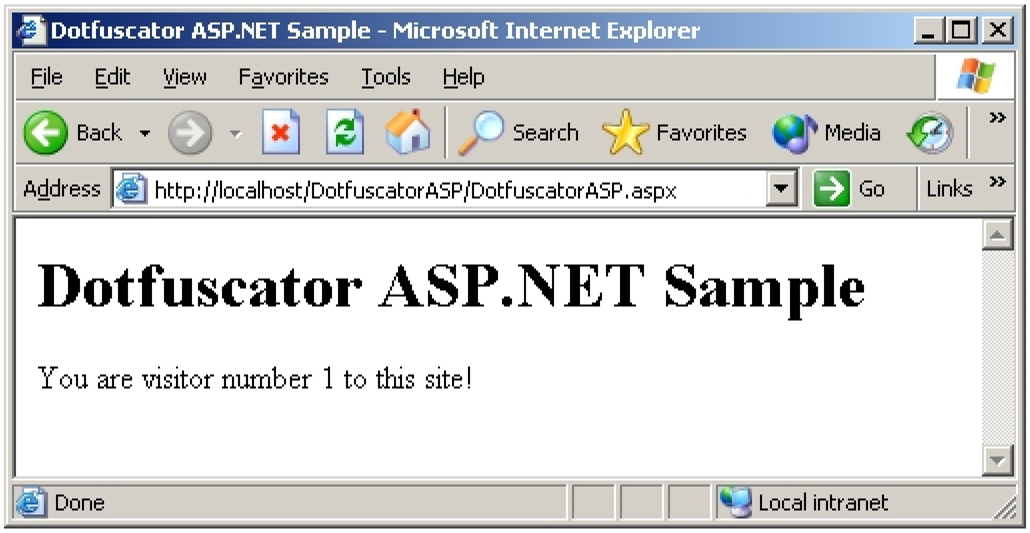
The asp.net sample assumes that the C# compiler (**csc.exe**) is reachable from your PATH environment variable. If you are using Visual Studio, you can make sure that it is in your path by executing the **Visual Studio Command Prompt** shortcut in your start menu.

From the command prompt, change your current directory to the directory containing the asp.net sample.

Build the application by executing **make.bat**. This batch file will invoke the Visual C# compiler to produce the output file - **DotfuscatorASP.dll**. This will be placed in a "bin" subdirectory of the current location. This is the default location that IIS will search for compiled code-behind assemblies.

2.7.4.4 Running the ASP.NET Sample

You can run the **asp.net** sample by directing your web browser to the **DotfuscatorASP.aspx** page in the virtual directory established for this sample. The sample has been configured correctly if you see the following:


Dotfuscator ASP.NET Sample - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/DotfuscatorASP/DotfuscatorASP.aspx Go Links
Done
Local intranet

Dotfuscator ASP.NET Sample

You are visitor number 1 to this site!

2.7.4.5 Examining the ASP.NET Sample Code

The aspx page itself is very simple:

ASPX Page

```
<%@ Page language="c#" Codebehind="DotfuscatorASP.aspx.cs" AutoEventWireup="false"
Inherits="DotfuscatorASP.Sample" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
    <HEAD>
        <title>Dotfuscator ASP.NET Sample</title>
    </HEAD>
    <body>
        <h1>Dotfuscator ASP.NET Sample</h1>
        <form id="SampleForm" method="post" runat="server">
            <p>You are visitor number&nbsp;<asp:Label id="CountLabel" runat="server"
/>&nbsp;to this site!</p>
        </form>
    </body>
</HTML>
```

All of the application logic resides in the **DotfuscatorASP.aspx.cs** code-behind file. Note that the aspx page references the class "**`DotfuscatorASP.Sample`**" and the field "**`CountLabel`**". These items are defined in the code behind. Since the aspx page is not processed until required by the web-server at runtime, these symbols must remain intact in the compiled code-behind assembly. Since they cannot be renamed, special steps must be taken to ensure that Dotfuscator will exclude them.

The class "**`DotfuscatorASP.Sample`**" has a static field, "**`PageLoadCount`**" that is incremented each time the page is loaded. This value is what is displayed in the *CountLabel* field and can be seen by the end-user in the browser. Note that this value is not persisted and will reset to the default value of **0** when the assembly is re-loaded by the web server.

2.7.4.6 Dotfuscating the ASP.NET Output

The asp.net sample contains a sample Dotfuscator configuration file that demonstrates using exclusion rules to handle items referenced in an aspx file that are define in the code-behind assembly. This file is named **asp_config.xml** and can be located in the same directory as the rest of the asp.net samples. The section of the file that excludes these references is:

ASP.NET Directory

```
<renaming>
  <excludelist>
   <type name="DotfuscatorASP.Sample">
    <field name="CountLabel"/>
   </type>
  </excludelist>
...
</renaming>
```

The **`<renaming>`** tag indicates that the exclusion rules contained within pertain specifically to identifier renaming, as opposed to other Dotfuscator features which can also be selectively turned on or off.

The **`<excludelist>`** tag defines a list of items which must be excluded from the renaming process. The **`<type name="DotfuscatorASP.Sample">`** tag instructs Dotfuscator to exclude the class name "**`DotfuscatorASP.Sample`**" from the renaming process. Note that this only refers to the class name itself. All methods of the "**`Tester`**" class are still eligible for renaming. This is required since the aspx file inherits from this class. The **`<field name="CountLabel"/>`** tag instructs Dotfuscator to exclude the server-side label control that displays the page load count.

Executing the run.bat file will run Dotfuscator with this configuration file. The output of this process is a **DotfuscatorASP.dll** assembly in the "**`output`**" subdirectory. This location can be controlled by modifying the following section in the configuration file:

Outut Directory

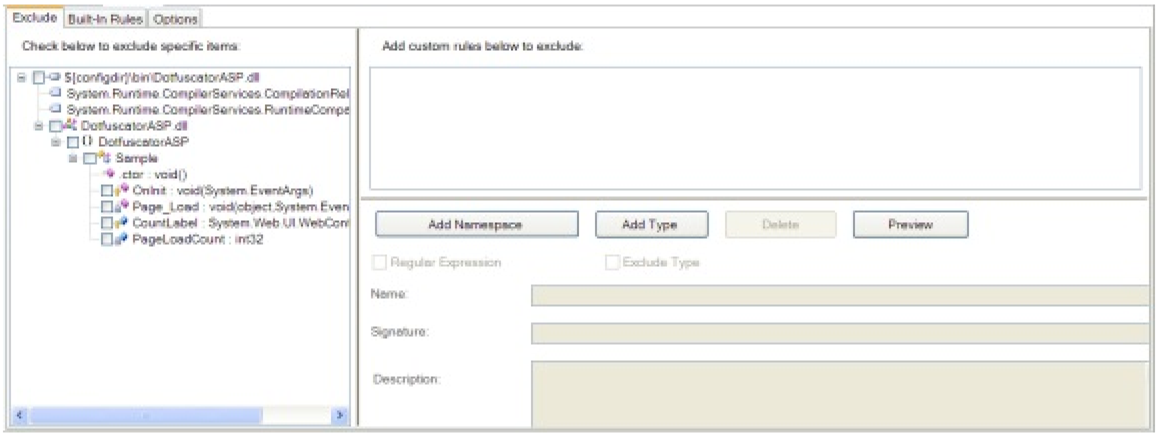
```
<output>
  <file dir="output" />
</output>
```

Executing the **deploy.bat** file will copy the obfuscated assembly into the bin directory that will be searched by the web server. This will overwrite the non-obfuscated assembly.

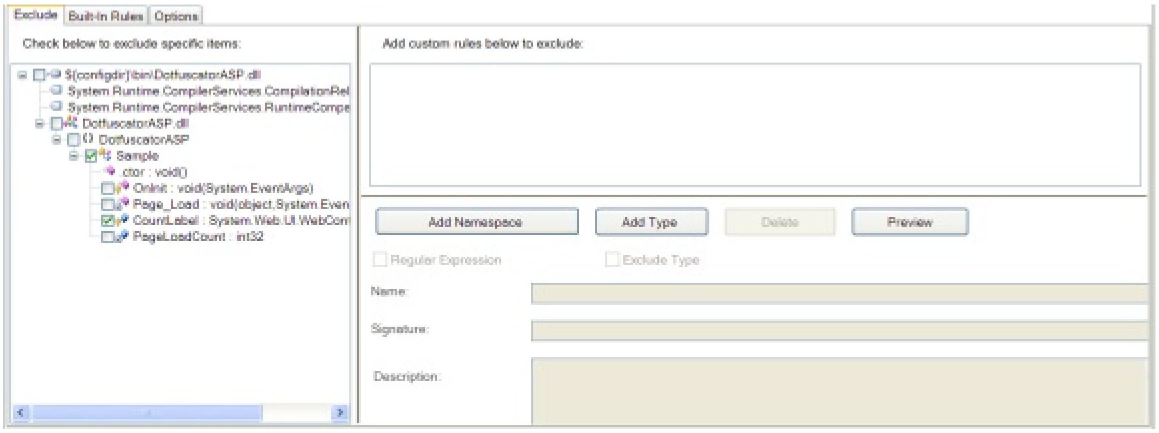
Visiting the website with the browser verifies that Dotfuscator correctly excluded the required items from the renaming process. Note that the load count has been reset to the default value indicating that IIS has reloaded the new assembly.

2.7.4.7 Configuring the ASP.NET Sample with the Graphical User Interface

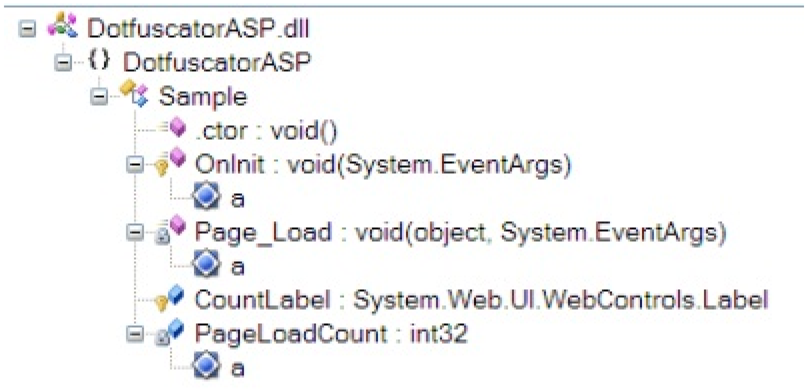
The Dotfuscator graphical interface provides a visual means to produce a configuration file. Items to be excluded from renaming can be specified using the *Rename* tab of the interface. Expanding the assembly node in the tree shows a graphical view of the application structure, including all namespaces, types, and methods.


Exclude
Built-In Rules
Options
Check below to exclude specific items:
${configdir}\bin\DotfuscatorASP.dll
System.Runtime.CompilerServices.CompilationRel
System.Runtime.CompilerServices.RuntimeCompa
DotfuscatorASP.dll
DotfuscatorASP
Sample
.ctor : void()
OnInit : void(System.EventArgs)
Page_Load : void(object,System.Even
CountLabel : System.Web.UI.WebCon
PageLoadCount : int32
Add custom rules below to exclude:
Add Namespace
Add Type
Delete
Preview
Regular Expression
Exclude Type
Name:
Signature:
Description:

Graphically generating a renaming exclusion list is a simple matter of checking the box next to the items to be excluded. In this case, we must exclude the type "**DotfuscatorASP.Sample**" and the field "**CountLabel**":


Exclude
Built-In Rules
Options
Check below to exclude specific items:
${configdir}\bin\DotfuscatorASP.dll
System.Runtime.CompilerServices.CompilationRel
System.Runtime.CompilerServices.RuntimeCompa
DotfuscatorASP.dll
DotfuscatorASP
Sample
.ctor : void()
OnInit : void(System.EventArgs)
Page_Load : void(object,System.Even
CountLabel : System.Web.UI.WebCon
PageLoadCount : int32
Add custom rules below to exclude:
Add Namespace
Add Type
Delete
Preview
Regular Expression
Exclude Type
Name:
Signature:
Description:

Building the project shows that the types were correctly excluded from the renaming process:



DotfuscatorASP.dll
{} DotfuscatorASP
Sample
.ctor : void()
OnInit : void(System.EventArgs)
a
Page_Load : void(object, System.EventArgs)
a
CountLabel : System.Web.UI.WebControls.Label
PageLoadCount : int32
a

2.7.4.8 Summary of the ASP.NET Sample

In order to Dotfuscate a web application which makes use of pre-compiled code-behind assemblies, care must be taken to make sure that any symbols referenced by aspx files are excluded from the renaming process. This includes any types that the aspx inherits from and any fields that it directly references by name. Everything else in the code-behind is eligible for obfuscation.

3 Index