

Logging Application Block Hands-On Lab for Enterprise Library



This walkthrough should act as your guide for learning about the Enterprise Library Logging Application Block and will allow you to practice employing its capabilities in various application contexts.

After completing this lab, you will be able to do the following:

- You will be able to use the Enterprise Library Logging Application Block to implement logging in an application.
- You will be able to create and use custom trace listeners.
- You will be able to create and use custom log formatters.

This hands-on lab includes the following three labs:

- [Lab 1: Add Logging to an Application](#)
- [Lab 2: Create and Use a Custom Trace Listener](#)
- [Lab 3: Create and Use a Custom Log Formatter](#)

The estimated completion for this lab is **30 minutes**.

Authors

These Hands-On Labs were produced by the following individuals:

- Product/Program Management: Grigori Melnik (Microsoft Corporation)
- Development: Chris Tavares (Microsoft Corporation), Nicolas Botto (Digit Factory), Olaf Conijn (Olaf Conijn BV), Fernando Simonazzi (Clarius Consulting), Erik Renaud (nVentive Inc.)
- Testing: Rick Carr (DCB Software Testing, Inc) plus everyone above
- Documentation: Alex Homer and RoAnn Corbisier (Microsoft Corporation) and Dennis DeWitt (Linda Werner & Associates Inc)

All of the Hands-On Labs use a simplified approach to object generation through Unity and the Enterprise Library container. The recommended approach when developing applications is to generate instances of Enterprise Library objects using dependency injection to inject instances of the required objects into your application classes, thereby realizing all of the advantages that this technique offers.

However, to simplify the examples and make it easier to see the code that uses the features of each of the Enterprise Library Application Blocks, the Hands-On Labs examples use the simpler approach to resolve Enterprise Library objects from the container by using the **GetInstance** method of the container service locator. You will see this demonstrated in each of the examples.

To learn more about using dependency injection to create instances of Enterprise Library objects, see the documentation installed with the Enterprise Library, or available on MSDN® at <http://msdn.microsoft.com/entlib/>.

Lab 1: Add Logging to an Application

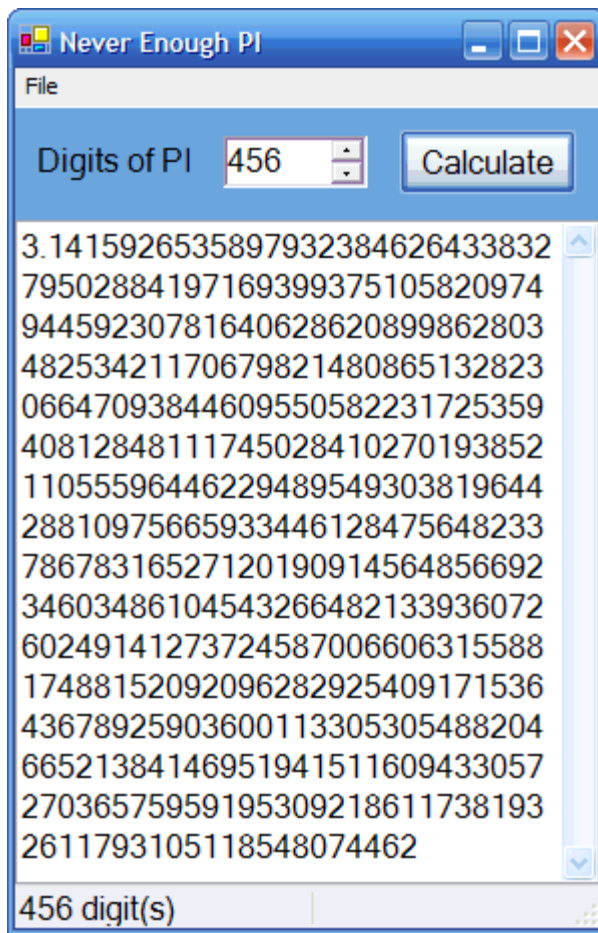
In this lab, you will add logging and tracing to an existing application. You will configure trace listeners via the Enterprise Library configuration tool.

To begin this exercise, open the EnoughPI.sln file located in the ex01\begin folder.

To learn about the application

1. Select the **Debug | Start Without Debugging** menu command to run the application.

The **EnoughPI** application calculates the digits of pi (π , the ratio of the circumference of a circle to its diameter). Enter your desired precision via the **NumericUpDown** control and click the **Calculate** button. Be prepared to wait if you want more than 500 digits of precision.



To add logging to the application

1. Select the **EnoughPI** project. Select the **Project | Add Reference** menu command. Select the **Browse** tab and select the following assemblies located in the Enterprise Library bin folder (typically C:\Program Files\Microsoft Enterprise Library 5.0 - April 2010\Bin):
 - Microsoft.Practices.EnterpriseLibrary.Logging.dll

- Microsoft.Practices.EnterpriseLibrary.Common.dll
- Microsoft.Practices.ServiceLocation.dll

2. Select the **Calc\Calculator.cs** file in Solution Explorer. Select the **View | Code** menu command.

Add the following namespace inclusions at the top of the file:

```
using Microsoft.Practices.EnterpriseLibrary.Logging;
using Microsoft.Practices.EnterpriseLibrary.Common.Configuration;
```

3. Add a private field to the class to hold a reference to the **LogWriter** instance that you will use to write log entries. Obtain an instance of the **LogWriter** class from the Enterprise Library container:

```
private LogWriter writer
    = EnterpriseLibraryContainer.Current.GetInstance<LogWriter>();
```

4. Log the calculation completion by adding the following highlighted code to the **OnCalculated** method in the Calculator.cs file.

```
protected void OnCalculated(CalculatedEventArgs args)
{
    // TODO: Log final result
    LogEntry log = new LogEntry();
    log.Message = string.Format("Calculated PI to {0} digits", args.Digits);
    log.Categories.Add(Category.General);
    log.Priority = Priority.Normal;

    writer.Write(log);

    if (Calculated != null)
        Calculated(this, args);
}
```

Create a new **LogEntry**, set parameters, then use the **Logger** to write the entry to one or more Tracelisters.

You have used constants for the **Category** and **Priority** rather than use hard-coded tags and integers (see Constants.cs in the EnoughPI.Logging project).

5. Log the calculation progress by adding the following code to the **OnCalculating** method in the Calculator.cs file.

```
protected void OnCalculating(CalculatingEventArgs args)
{
    // TODO: Log progress
    writer.Write(
        string.Format("Calculating next 9 digits from {0}", args.StartingAt),
        Category.General,
        Priority.Low
    );
}
```

```

    );

    if (Calculating != null)
        Calculating(this, args);

    if (args.Cancel == true)
    {
        // TODO: Log cancellation
        writer.Write("Calculation cancelled by user!",
            Category.General, Priority.High);
    }
}

```

You have used an overload of the **Write** method on the **Logger** class as a shortcut to creating a **LogEntry**.

6. Log calculation exceptions by adding the following code to the **OnCalculatorException** method in the Calculator.cs file.

```

protected void OnCalculatorException(CalculatorExceptionEventArgs args)
{
    // TODO: Log exception
    if (!(args.Exception is ConfigurationErrorsException))
    {
        writer.Write(args.Exception, Category.General, Priority.High);
    }

    if (CalculatorException != null)
        CalculatorException(this, args);
}

```

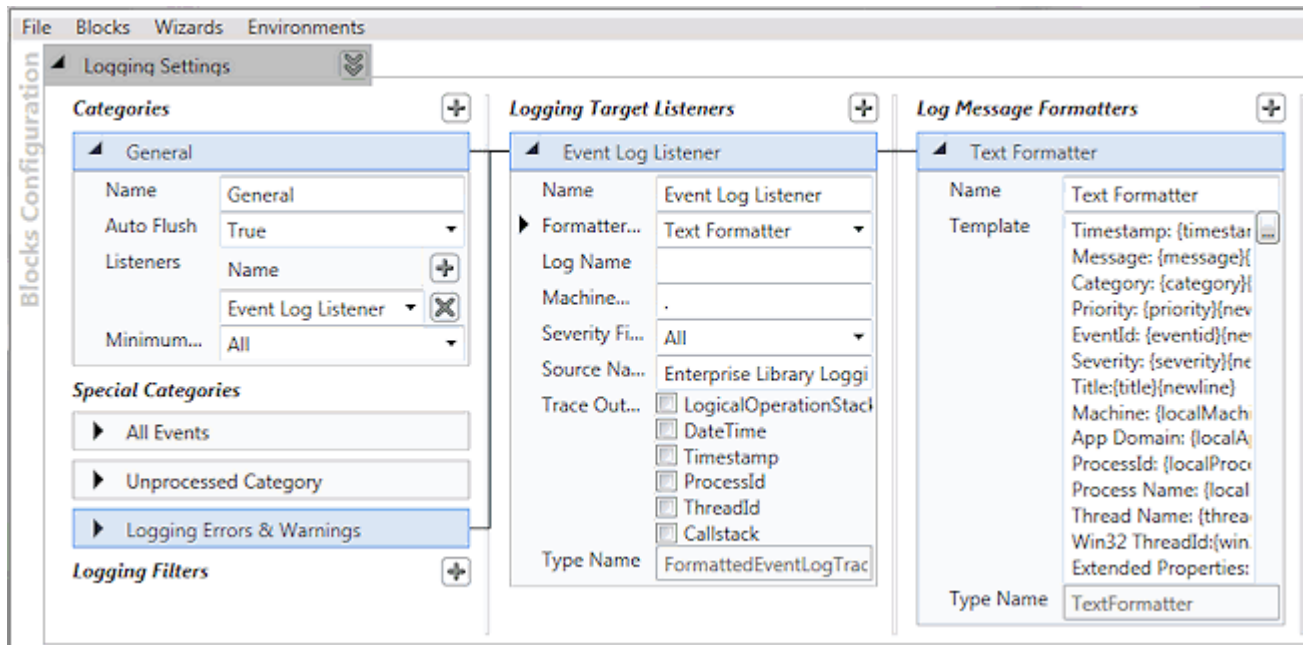
You must test that the exception type is not a **ConfigurationErrorsException** as you would not be able to use the **Logger** if it has not been correctly configured.

You would normally use the Enterprise Library Exception Handling Application Block to create a consistent strategy for processing exceptions.

To configure the application

1. You will use the Enterprise Library configuration tool to configure your application. Open the application configuration in the configuration editor using one of these approaches:
 - Start the appropriate version of the stand-alone tool from the Windows® Start menu (select **All Programs | Microsoft patterns & practices | Enterprise Library 5.0 | Enterprise Library Configuration**) and open the App.config file.
 - In Visual Studio®, right-click on the App.config file in Solution Explorer and click **Edit Enterprise Library V5 Configuration**.


2. Open the **Blocks** menu and click **Add Logging Settings**. This automatically adds the Logging Application Block to your configuration with the default configuration of a category named **General** that writes log entries to Windows Event Log using an event log listener and a text log message formatter.
3. Click the expander arrows to the left of the **General**, **Event Log Listener**, and **Text Formatter** titles of the default items added to the configuration. This reveals the property editing section for each item (see the screen shot below).



Categories are simply text tags that you may apply to your log events to group them. The **General** category has one (trace) listener reference named **Event Log Listener** that refers to the event log listener in the **Logging Target Listeners** column.

New categories may be added by clicking the plus-sign icon in the **Categories** column and selecting **Add Category**. A category may have many trace listener references (though none can be repeated), and a trace listener may be referenced by many categories.

4. Change the **Source Name** property of the **Event Log Listener** to **EnoughPI**, as shown in the screen shot that follows. This is the name you will see for log entries when they appear in the Windows Event Log.

Logging Target Listeners 

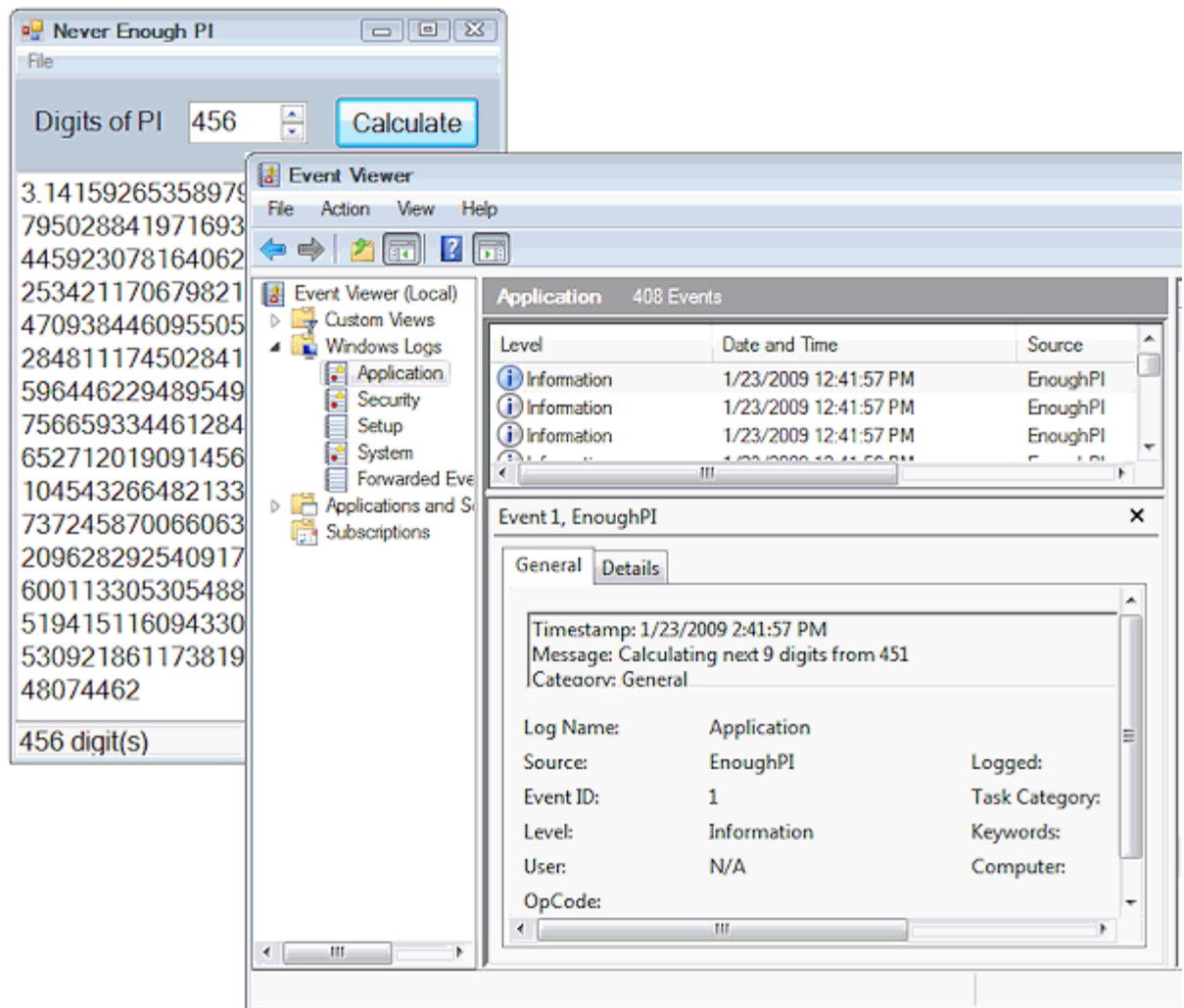
Event Log Listener	
Name	Event Log Listener
▶ Formatter Name	Text Formatter ▼
Log Name	
Machine Name	.
Severity Filter	All ▼
Source Name	EnoughPI
Trace Output Options	<input type="checkbox"/> LogicalOperationStack <input type="checkbox"/> DateTime <input type="checkbox"/> Timestamp <input type="checkbox"/> ProcessId <input type="checkbox"/> ThreadId <input type="checkbox"/> Callstack
Type Name	FormattedEventLogTraceListene

Sources must be registered with the event log. The event log trace listener automatically registers a new source the first time it is used, but this requires administrative privileges, so running the application as an administrator the first time is necessary.

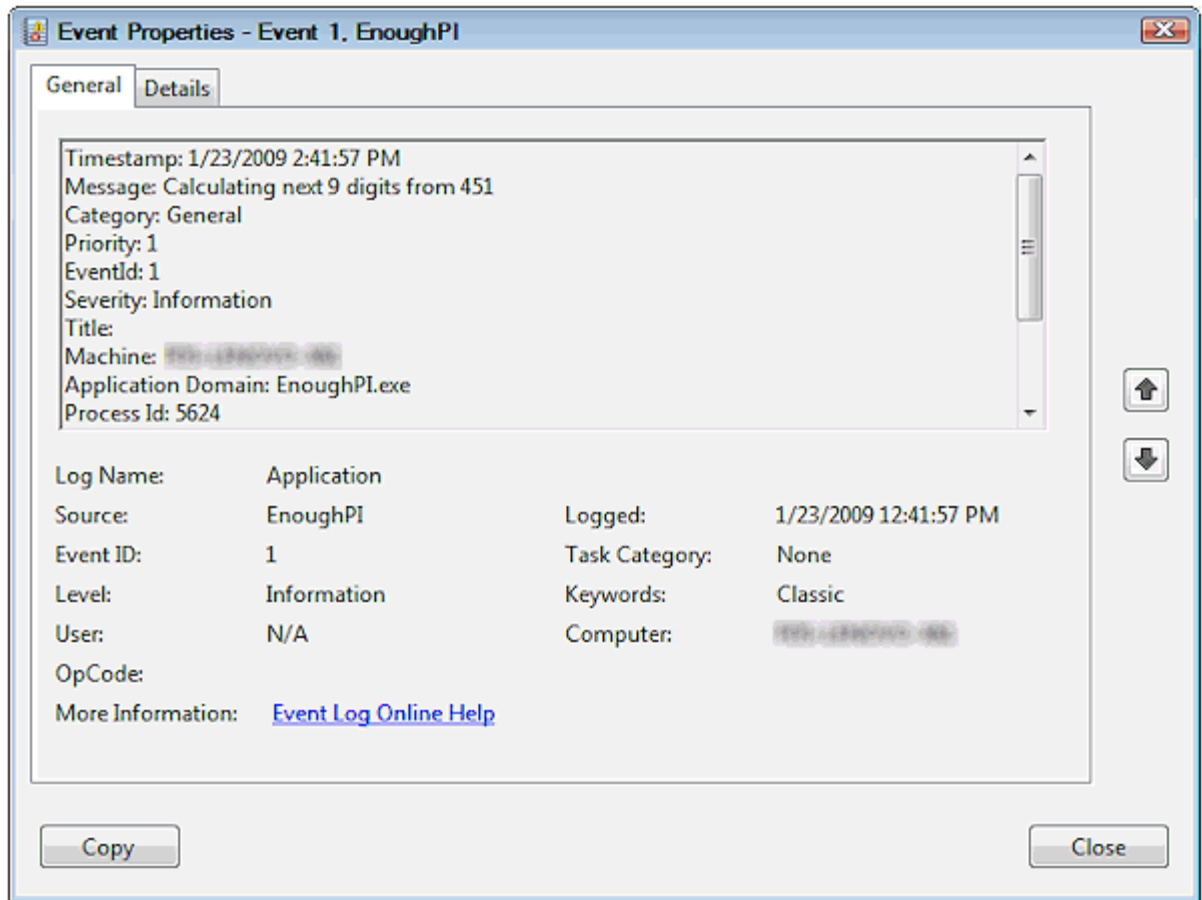
5. Save the application configuration.

To run the application

1. Select the **Debug | Start Without Debugging** menu command to run the application. Enter your desired precision and click the **Calculate** button. As mentioned above, the first time you run the application, you'll need to have administrative privileges to get the event source registered.
2. Run the event viewer. From the Windows **Start** menu select **Administrative Tools | Event Viewer**. View the application log for messages from the **EnoughPI** source, as shown below.



3. Double-click a log entry to view the formatted log message, as the following screen shot illustrates.



4. Exit the application.

Often, you would like to time the execution of sections of your application. The Logging Application Block includes tracing, which allows you to book-end a section of code and log the execution time.

To add tracing to the application

1. Select the Calc\Calculator.cs file in the Solution Explorer. Select the **View | Code** menu command.
2. Add a private field to the class to hold a reference to the **TraceManager** instance that you will use to trace execution. Obtain an instance of the **TraceManager** class from the Enterprise Library container:

```
private TraceManager traceMgr  
    = EnterpriseLibraryContainer.Current.GetInstance<TraceManager>();
```

3. Locate the **Calculate** method and add the following highlighted code to create a new **Tracer** instance that wraps the code that performs the calculation:

```
public string Calculate(int digits)  
{  
    StringBuilder pi = new StringBuilder("3", digits + 2);
```

```

string result = null;

try
{
    if (digits > 0)
    {
        // TODO: Add Tracing around the calculation
        using (Tracer trace = traceMgr.StartTrace(Category.Trace))
        {
            pi.Append(".");
            for (int i = 0; i < digits; i += 9)
            {
                CalculatingEventArgs args;
                args = new CalculatingEventArgs(pi.ToString(), i+1);
                OnCalculating(args);

                // Break out if cancelled
                if (args.Cancel == true) break;

                // Calculate next 9 digits
                int nineDigits = NineDigitsOfPi.StartingAt(i+1);
                int digitCount = Math.Min(digits - i, 9);
                string ds = string.Format("{0:D9}", nineDigits);
                pi.Append(ds.Substring(0, digitCount));
            }
        }
    }

    result = pi.ToString();

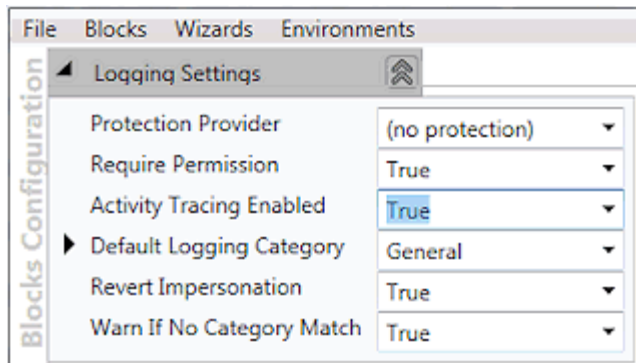
    // Tell the world I've finished!
    OnCalculated(new CalculatedEventArgs(result));
}
catch (Exception ex)
{
    // Tell the world I've crashed!
    OnCalculatorException(new CalculatorExceptionEventArgs(ex));
}

return result;}

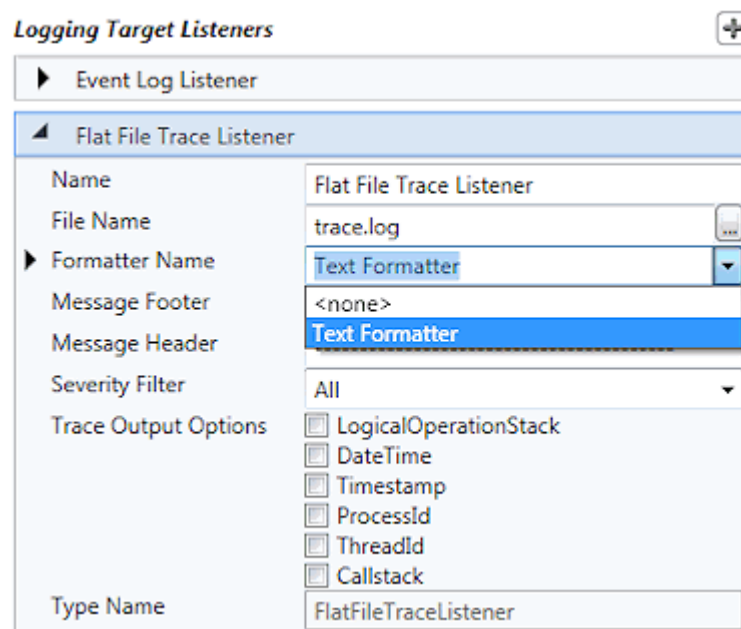
```

The tracer will stop timing, and log its end trace message, when it is disposed. The **using** block guarantees that **Dispose** will be called on the tracer at the end of the block. Allowing the garbage collector to dispose of the tracer will result in incorrect timings.

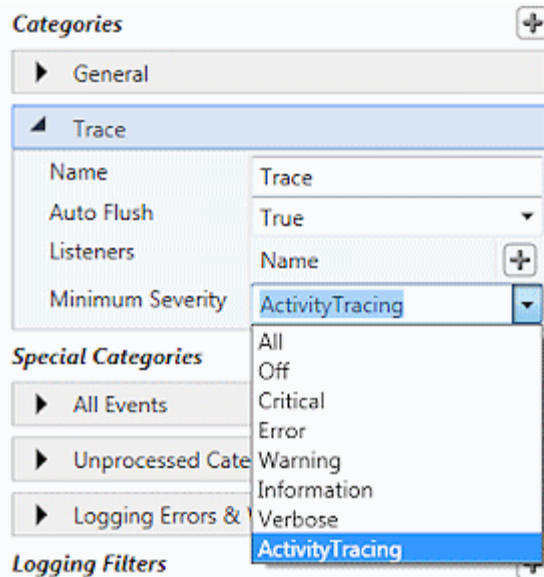
4. Open your App.config file in the Enterprise Library configuration editor.
5. Click the chevron arrow icon to the right of the **Logging Settings** title to show the properties for this configuration section, and confirm that **Activity Tracing Enabled** is set to **True**.



6. Add a new trace listener that will write its output to a disk file. Click the plus-sign icon in the **Logging Target Listeners** column, point to **Add Logging Target Listeners**, and select **Add Flat File Trace Listener**.
7. Select the existing text formatter as the formatter to use to format the log entries for the new flat file trace listener by selecting **Text Formatter** in the drop-down list for the **Formatter Name** property, as you see here.

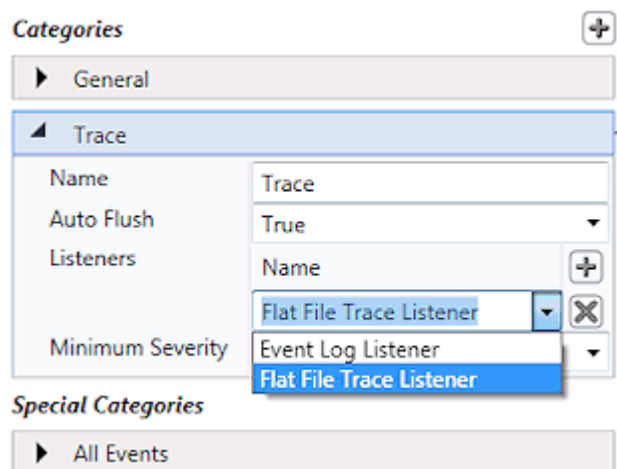


8. Add a new Trace category. Click the plus-sign icon in the **Categories** column, and click **Add Category**.
9. Change the **Name** property of the new category item to **Trace**, and select **ActivityTracing** in the drop-down list for the **Minimum Severity** property:



The category name **Trace** was used in our code (see Constants.cs in the EnoughPI.Logging project). Using the **ActivityTracing** source level will restrict the trace logging to the start and end log entries only.

- Now you must add the flat file trace listener that you previously configured to the new category named **Trace**. Click the plus-sign icon next to the **Listeners** property of the **Trace** category, and then select **Flat File Trace Listener** in the drop-down list of available trace listeners.



- Save the application configuration.

To run the application with tracing

- Select the **Debug | Start Without Debugging** menu command to run the application. Enter your desired precision and click the **Calculate** button.
- You may view the elapsed time for the trace in the trace.log file, which you will find in your ex01\begin\EnoughPI\bin\Debug\ folder.

```

-----
Timestamp: 13/12/2005 6:08:01 AM
Message: Start Trace: Activity '8c07ce3b-235b-4a51-bdcc-83a5997c989e' in
method 'Calculate' at 71661842482 ticks
Category: Trace
Priority: 5
EventId: 1
Severity: Start
Title:TracerEnter
Machine: #####
Application Domain: EnoughPI.exe
Process Id: 6016
Process Name: C:\Program Files\Microsoft Enterprise
Library\labs\cs\Logging\exercises\ex01\begin\EnoughPI\bin\EnoughPI.exe
Win32 Thread Id: 6092
Thread Name:
Extended Properties:
-----

Timestamp: 13/12/2005 6:08:01 AM
Message: End Trace: Activity '8c07ce3b-235b-4a51-bdcc-83a5997c989e' in method
'Calculate' at 71662624219 ticks (elapsed time: 0.218 seconds)
Category: Trace
Priority: 5
EventId: 1
Severity: Stop
Title:TracerExit
Machine: #####
Application Domain: EnoughPI.exe
Process Id: 6016
Process Name: C:\Program Files\Microsoft Enterprise
Library\labs\cs\Logging\exercises\ex01\begin\EnoughPI\bin\EnoughPI.exe
Win32 Thread Id: 6092
Thread Name:
Extended Properties:
-----

```

Your file should look similar to the above output, though will have different values for the GUID, machine name, and other process-specific details.

3. Close the application and Visual Studio.

To verify that you have completed the exercise correctly, you can use the solution provided in the ex01\end folder.

Lab 2: Create and Use a Custom Trace Listener

In this lab, you will build a custom Trace Listener to send formatted log entries to the Console standard output. You will then add this new Trace Listener to the EnoughPI application and monitor the log entries in real-time.

To begin this exercise, open the EnoughPI.sln file located in the ex02\begin folder.

To create a custom Trace Listener

1. Select the **EnoughPI.Logging** project. Select the **Project | Add Reference** menu command. Select the **Browse** tab and select the following assemblies located in the Enterprise Library **bin** folder (typically C:\Program Files\Microsoft Enterprise Library 5.0\Bin):
 - Microsoft.Practices.EnterpriseLibrary.Common.dll
 - Microsoft.Practices.EnterpriseLibrary.Logging.dll
 - Microsoft.Practices.ServiceLocation.dll
2. Select the **TraceListeners\ConsoleTraceListener.cs** file in the Solution Explorer. Select the **View | Code** menu command. Add the following namespaces:

```
using Microsoft.Practices.EnterpriseLibrary.Common.Configuration;
using Microsoft.Practices.EnterpriseLibrary.Logging;
using Microsoft.Practices.EnterpriseLibrary.Logging.Configuration;
using Microsoft.Practices.EnterpriseLibrary.Logging.Formatters;
using Microsoft.Practices.EnterpriseLibrary.Logging.TraceListeners;
```

3. Add the following highlighted code to the **ConsoleTraceListener** class.

```
[ConfigurationElementType(typeof(CustomTraceListenerData))]
public class ConsoleTraceListener : CustomTraceListener
{
    public ConsoleTraceListener()
        : base()
    {
    }

    public override void TraceData(TraceEventCache eventCache,
        string source, TraceEventType eventType, int id, object data)
    {
        if (data is LogEntry && this.Formatter != null)
        {
            this.WriteLine(this.Formatter.Format(data as LogEntry));
        }
        else
        {
            this.WriteLine(data.ToString());
        }
    }
}
```

```

public override void Write(string message)
{
    Console.Write(message);
}

public override void WriteLine(string message)
{
    // Delimit each message
    Console.WriteLine((string)this.Attributes["delimiter"]);

    // Write formatted message
    Console.WriteLine(message);
}
}

```

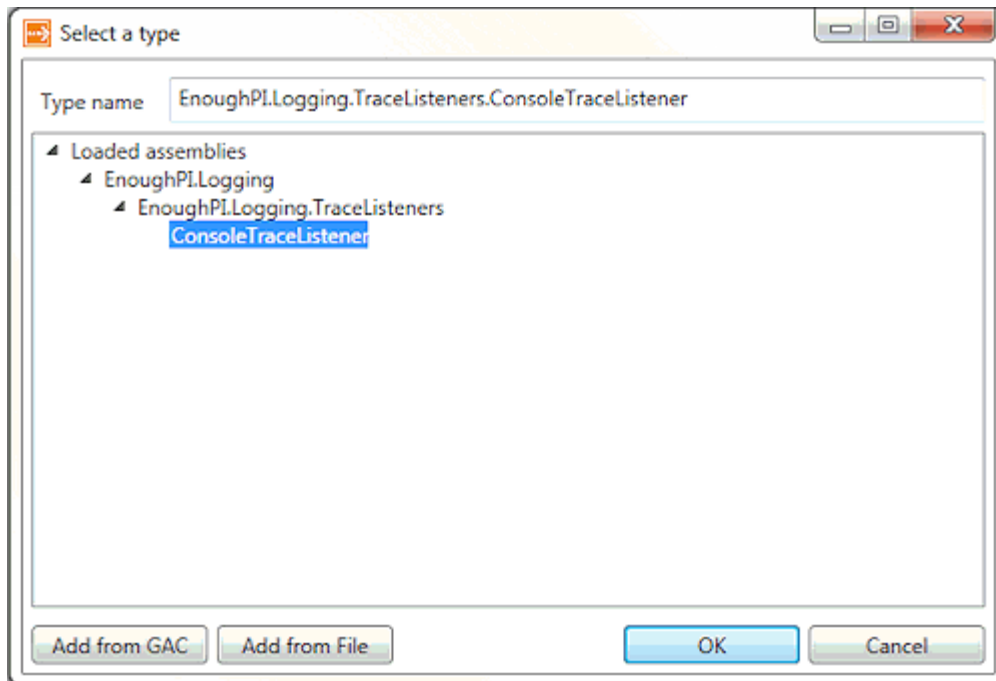
The base class is **CustomTraceListener**, which mandates that you override two abstract methods: **Write(string message)** and **WriteLine(string message)**. However, to format the message we need to override the **TraceData** method.

The **ConsoleTraceListener** is expecting a parameter, **delimiter**, as part of the listener configuration.

4. Select the **Build | Build Solution** menu command, to compile the complete solution.

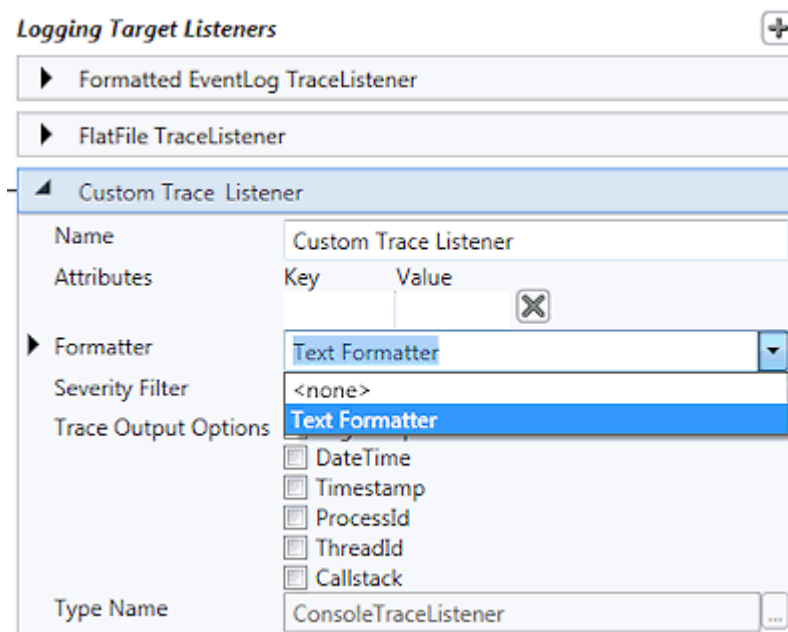
To use a custom Trace Listener

1. Open your App.config file in the Enterprise Library configuration editor, and click the expander arrow in the **Logging Settings** section to show the configuration for the Logging Application Block.
2. Click the plus-sign icon in the **Logging Target Listeners** column, point to **Add Logging Target Listeners**, and select **Add Custom Trace Listener**.
3. In the type selector dialog that appears, click **Add from File**, navigate to the EnoughPI.Logging\bin\Debug folder, select **EnoughPI.Logging.dll**, and click **Open**. Then, back in the type selector dialog, expand the entries in the list, select **ConsoleTraceListener**, and then click **OK**.



The type selector for a custom trace listener lists classes which inherit from **CustomTraceListener** and have a **ConfigurationElementType** of **CustomTraceListenerData**.

4. The custom trace listener appears in the configuration, where you can set its properties. Change the **Name** property to **Custom Trace Listener**. In drop-down list for the **Formatter** property, select **Text Formatter** as the formatter to use for this trace listener.



5. In the **Attributes** property section, you enter the name/value pairs that will provide configuration settings for the custom trace listener. Enter **delimiter** as the **Key**, and a string of

hyphens as the **Value**. As you add an entry, the configuration tool automatically adds a new blank row.

Logging Target Listeners (+)

- ▶ Formatted EventLog TraceListener
- ▶ FlatFile TraceListener
- ▲ Custom Trace Listener
 - Name: Custom Trace Listener
 - Attributes:

Key	Value
delimiter	-----
 - ▶ Formatter: Text Formatter
 - Severity Filter: All
 - Trace Output Options:
 - ☐ LogicalOperationStack
 - ☐ DateTime
 - ☐ Timestamp
 - ☐ ProcessId
 - ☐ ThreadId
 - ☐ Callstack
 - Type Name: ConsoleTraceListener

You will remember your **ConsoleTraceListener** is expecting a parameter named **delimiter**, which is printed before each formatted log entry is written to the console.

- Now you must add your custom trace listener to the required logging category. Click the plus-sign icon next to the **Listeners** property of the **General** category (which already references the event log trace listener) and select **Custom Trace Listener** in the drop-down list of available trace listeners, as shown below.

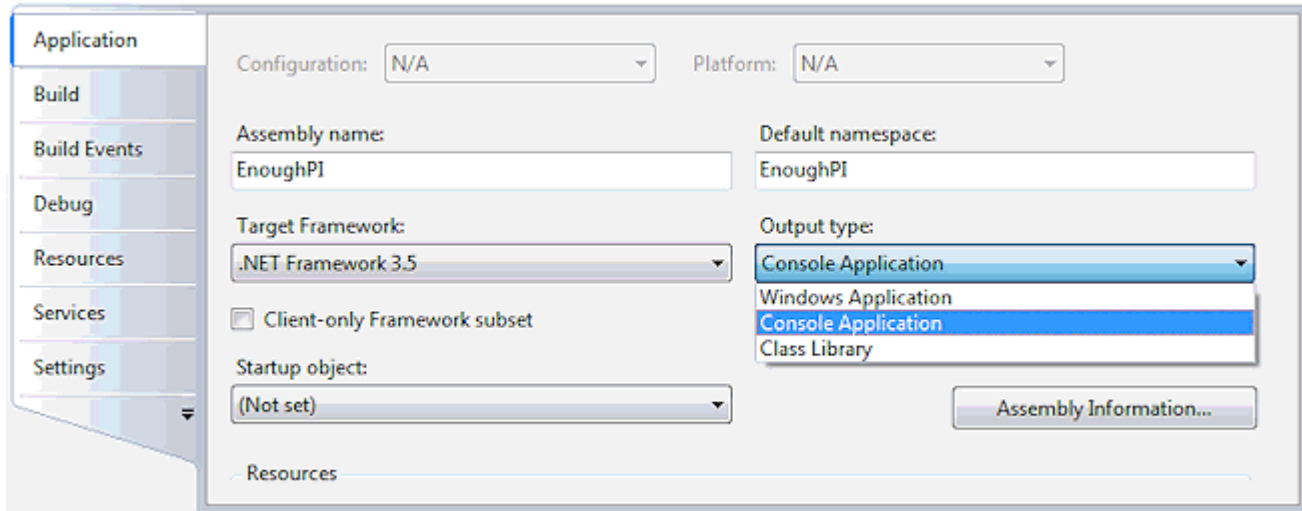
Categories (+)

- ▲ General
 - Name: General
 - Auto Flush: True
 - Listeners:
 - Name: +
 - Event Log Trace Listener
 - Custom Trace Listener
 - Flat File TraceListener
 - Minimum Severity: Custom Trace Listener
- ▶ Trace

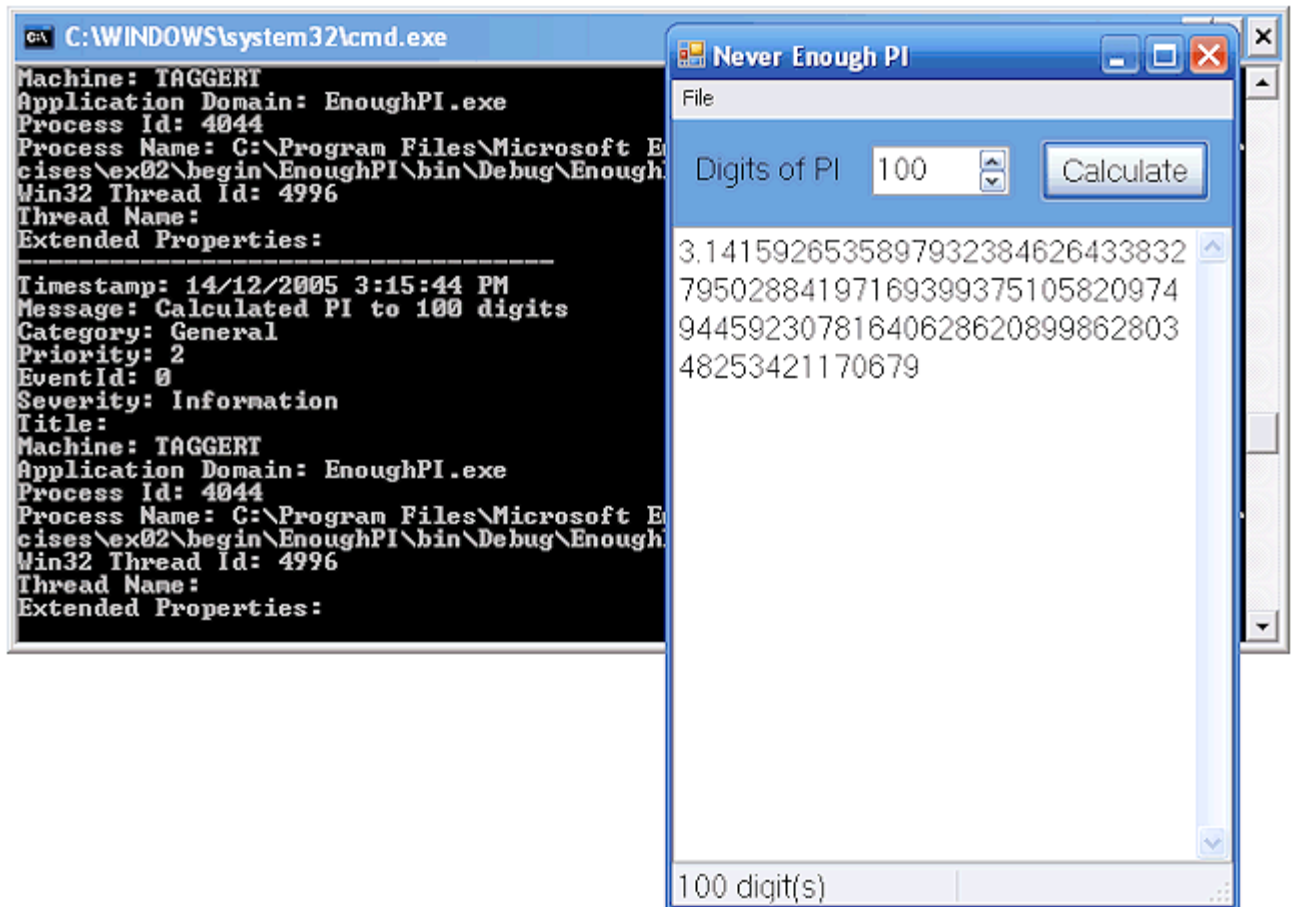
- Select the **File | Save** menu command.

To view the Trace Listener output

1. Select the **EnoughPI** project. Select the **Project | EnoughPI Properties...** menu command, select the **Application** tab, and set the following:
 - **Output type = Console Application**



2. Select the **File | Save All** menu command.
3. Select the **Debug | Start Without Debugging** menu command to run the application. Enter your desired precision and click the **Calculate** button. The log entries will be displayed in the application's console window, as you see here.



To verify that you have completed the exercise correctly, you can use the solution provided in the ex02\end folder.

Lab 3: Create and Use a Custom Log Formatter

In this lab, you will add a custom log formatter to a logging application.

To begin this exercise, open the EnoughPI.sln file located in the ex03\begin folder.

To create a custom log formatter

1. Select the **Formatters\XmlFormatter.cs** file in the Solution Explorer. Select the **View | Code** menu command. Add the following namespaces:

```
using Microsoft.Practices.EnterpriseLibrary.Common.Configuration;
using Microsoft.Practices.EnterpriseLibrary.Logging;
using Microsoft.Practices.EnterpriseLibrary.Logging.Configuration;
using Microsoft.Practices.EnterpriseLibrary.Logging.Formatters;
```

2. Add the following highlighted code to the **XmlFormatter** class.

```
[ConfigurationElementType(typeof(CustomFormatterData))]
public class XmlFormatter : LogFormatter
{
    private NameValueCollection Attributes = null;

    public XmlFormatter(NameValueCollection attributes)
    {
        this.Attributes = attributes;
    }

    public override string Format(LogEntry log)
    {
        string prefix = this.Attributes["prefix"];
        string ns = this.Attributes["namespace"];

        using (StringWriter s = new StringWriter())
        {
            XmlTextWriter w = new XmlTextWriter(s);
            w.Formatting = Formatting.Indented;
            w.Indentation = 2;
            w.WriteStartDocument(true);
            w.WriteStartElement(prefix, "logEntry", ns);
            w.WriteAttributeString("Priority", ns,
                log.Priority.ToString(CultureInfo.InvariantCulture));
            w.WriteElementString("Timestamp", ns, log.TimeStampString);
            w.WriteElementString("Message", ns, log.Message);
            w.WriteElementString("EventId", ns,
                log.EventId.ToString(CultureInfo.InvariantCulture));
            w.WriteElementString("Severity", ns, log.Severity.ToString());
            w.WriteElementString("Title", ns, log.Title);
            w.WriteElementString("Machine", ns, log.MachineName);
            w.WriteElementString("AppDomain", ns, log.AppDomainName);
            w.WriteElementString("ProcessId", ns, log.ProcessId);
        }
    }
}
```

```

        w.WriteElementString("ProcessName", ns, log.ProcessName);
        w.WriteElementString("Win32ThreadId", ns, log.Win32ThreadId);
        w.WriteElementString("ThreadName", ns, log.ManagedThreadName);
        w.WriteEndElement();
        w.WriteEndDocument();

        return s.ToString();
    }
}

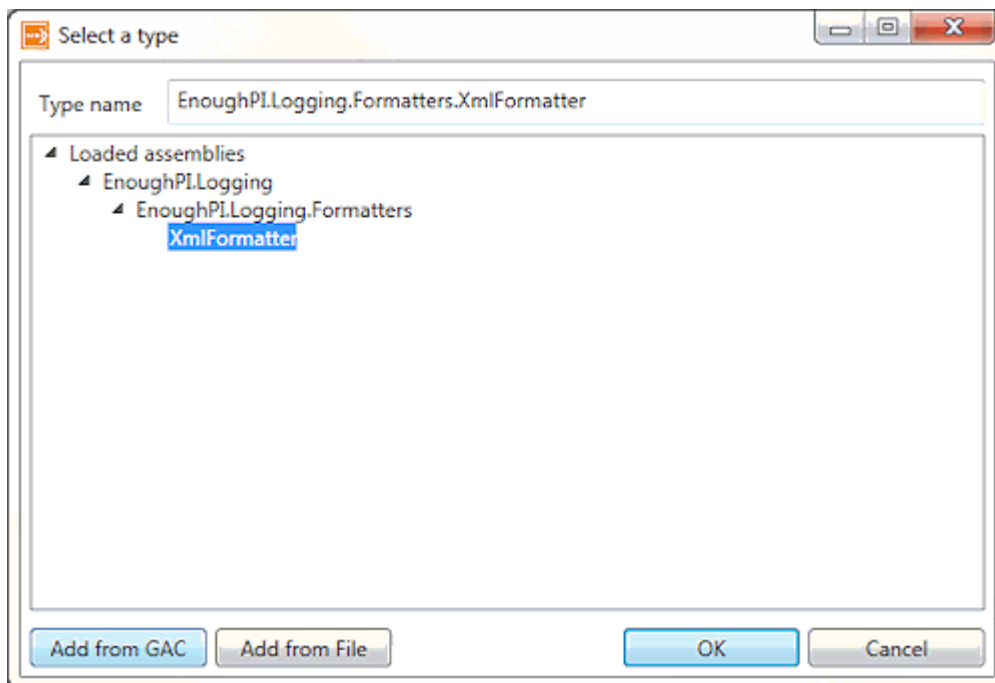
```

The log entry will be formatted as XML. Two parameters are expected (**prefix** and **namespace**).

3. Select **Build | Build Solution** to compile the complete solution.

To use a custom log formatter

1. Open your App.config file in the Enterprise Library configuration editor, and click the expander arrow in the **Logging Settings** section to show the configuration for the Logging Application Block.
2. Click the plus-sign icon in the **Log Message Formatters** column, point to **Add Log Message Formatters**, and select **Add Custom Log Message Formatter**.
3. In the type selector dialog that appears, click **Add from File**, navigate to the EnoughPI.Logging\bin\Debug folder, select **EnoughPI.Logging.dll**, and click **Open**. Then, back in the type selector dialog, expand the entries in the list, select **XmlFormatter**, as shown in the screen shot, and then click **OK**.



Make sure you are loading the assembly from the appropriate output folder (typically Logging\exercises\ex03\begin\EnoughPI.Logging\bin\Debug).

The type selector for a custom log formatter lists classes that inherit from **LogFormatter** and have a **ConfigurationElementType** of **CustomFormatterData**.

4. The custom log formatter appears in the configuration, where you can set its properties. Change the **Name** property to **Xml Formatter** (with a space between the words, as shown below).

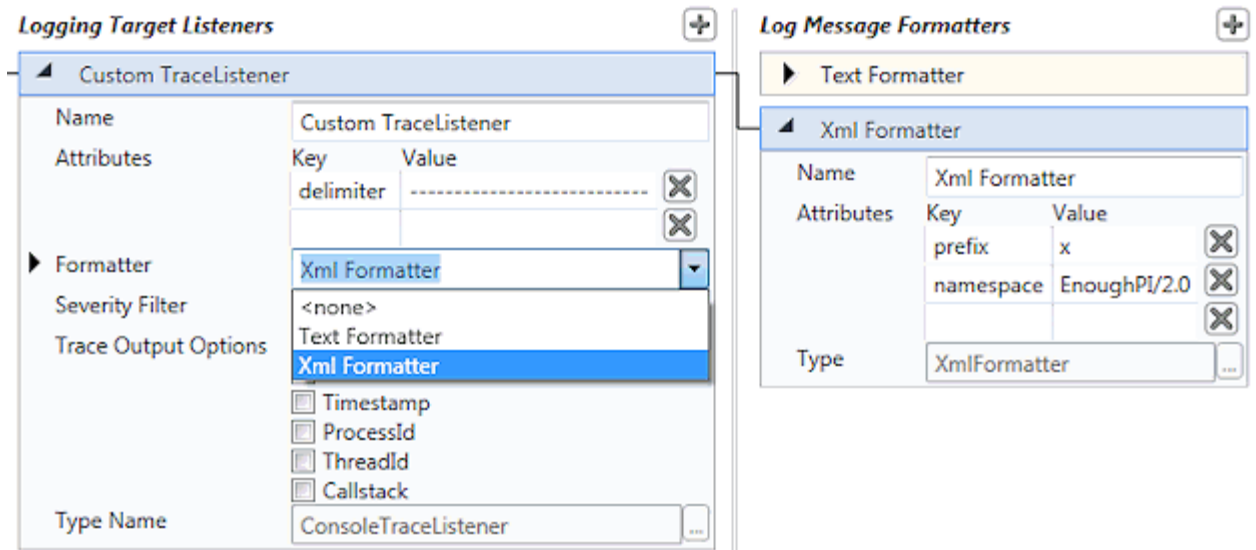
The screenshot shows the 'Log Message Formatters' window. The 'Text Formatter' is expanded, and the 'Xml Formatter' is selected. The 'Name' property is set to 'Xml Formatter'. The 'Attributes' section is empty. The 'Type' property is set to 'XmlFormatter'.

5. In the **Attributes** property section, you enter the name/value pairs that will provide configuration settings for the custom log formatter. Enter **prefix** as the **Key**, and **x** as the **Value**. As you add an entry, the configuration tool automatically adds a new blank row. In the next row, add **namespace** as the **Key**, and **EnoughPI/2.0** as the **Value** (see below).

The screenshot shows the 'Log Message Formatters' window. The 'Text Formatter' is expanded, and the 'Xml Formatter' is selected. The 'Name' property is set to 'Xml Formatter'. The 'Attributes' section contains two entries: 'prefix' with value 'x' and 'namespace' with value 'EnoughPI/2.0'. The 'Type' property is set to 'XmlFormatter'.

You will remember your **XmlFormatter** is expecting two parameters, the prefix and namespace for the XML document.

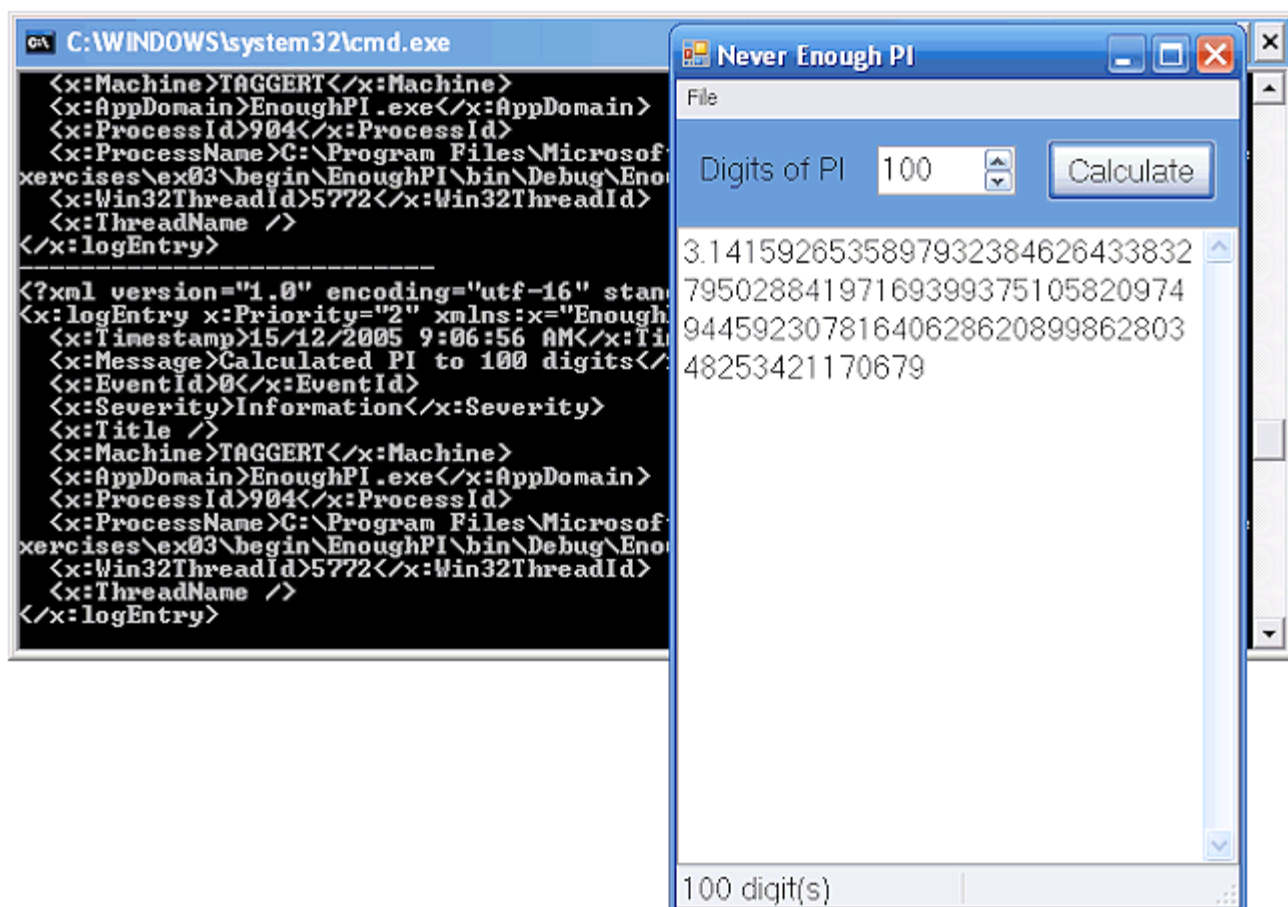
6. Now you must specify your custom log formatter as the formatter that the custom trace listener you added in the previous exercise will use to format the log entry. In the drop-down list of available formatters in the **Formatter** property of the **Custom Trace Listener**, select your **Xml Formatter**, as illustrated below.



7. Select the **File | Save** menu command.

To view the Formatter output

1. Select the **Debug | Start Without Debugging** menu command to run the application. Enter your desired precision and click the **Calculate** button. The log entries will be displayed as XML in the application's console window.



To verify that you have completed the exercise correctly, you can use the solution provided in the ex03\end folder.



patterns & practices
proven practices for predictable results

Copyright

This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes. You may modify this document for your internal, reference purposes

© 2010 Microsoft. All rights reserved.

Microsoft, MSDN, Visual Studio, and Windows are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.