Logging Application Block Hands-On Lab for Enterprise Library

patterns & practices
proven practices for predictable results

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 use the Logging Application Block to log messages asynchronously.
- You will be able to dynamically reconfigure logging while the application is running.
- 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 five labs:

The estimated completion for this lab is **45 minutes**.

Authors

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

- Program Management: Grigori Melnik and Madalyn Parker (Microsoft Corporation)
- Development/Testing: Fernando Simonazzi (Clarius Consulting), Chris Tavares, Grigori Melnik, Mike Sampson and Cim Ryan (Microsoft Corporation), Erik Renaud (nVentive Inc.), Mariano Grande (Digit Factory), and Naveen Pitipornvivat (Adecco)
- Documentation: Fernando Simonazzi (Clarius Consulting), Grigori Melnik, Alex Homer, Nelly Delgado and RoAnn Corbisier (Microsoft Corporation)

Lab 1: Add Logging to an Application

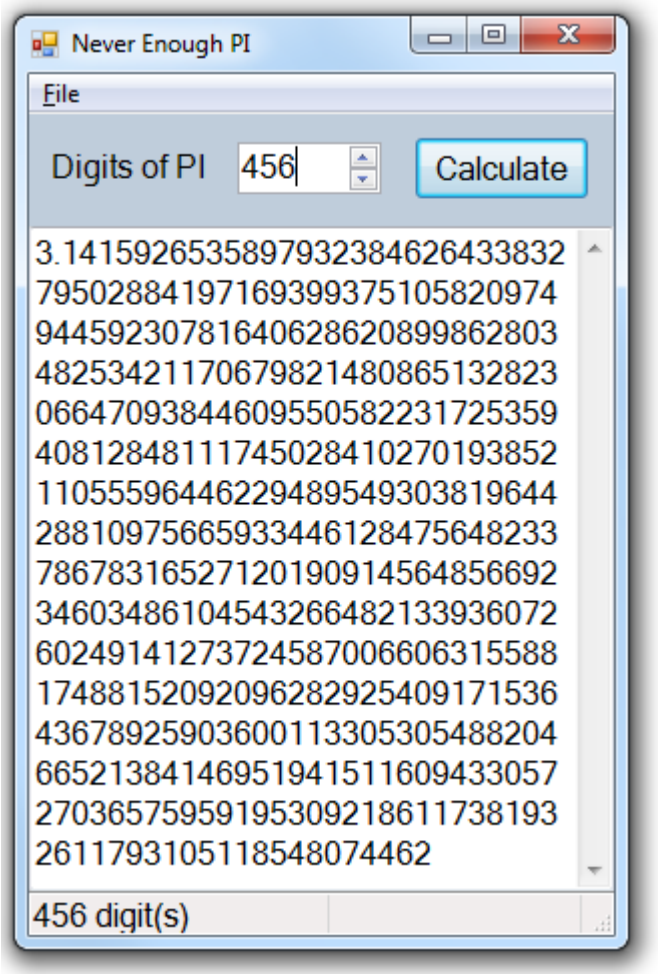
In this lab, you will add logging and tracing to an existing application. To begin this exercise, open the EnoughPI.sln file located in the ex01\begin folder.

> **Note:** This exercise involves writing to the event log. The event log trace listener used in this application automatically registers a new event source, but this requires you to run Visual Studio with elevated permissions. Please run Visual Studio as Administrator for this lab.

To learn about the application

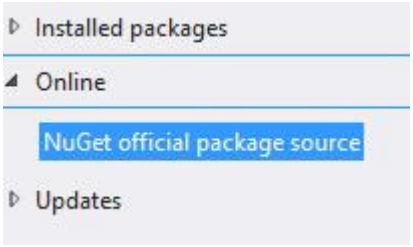
1. Select the **Debug | Start Without Debugging** menu command to run the application.

   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.

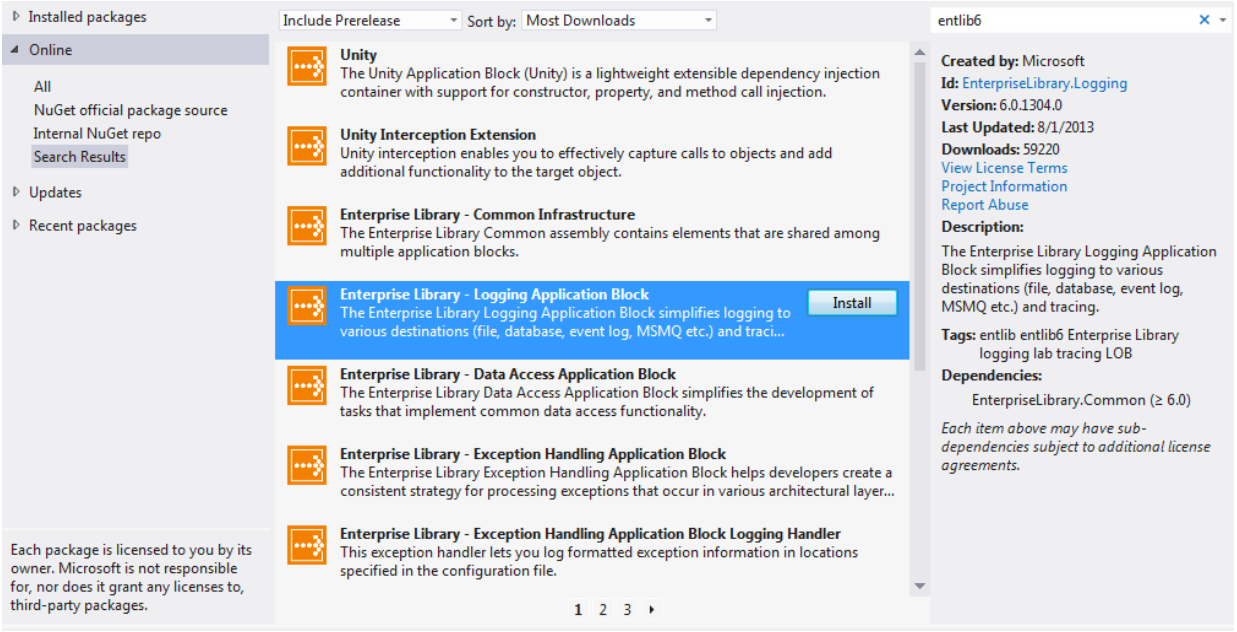


To add logging to the application

1. Select the **EnoughPI** project. Select the **Project | Manage NuGet Packages** menu command or right-click on the References in the Solution Explorer.
2. Select the "Online" option to view NuGet packages available online.



3. Search for **EntLib6** in the search bar. Select **Enterprise Library – Logging Application Block** and click install.



4. Click **Accept** on the License Acceptance window that pops up.
5. Select the **EntryPoint.cs** file in Solution Explorer and view it.

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

```
using System.Diagnostics;
using EnoughPI.Logging;
using Microsoft.Practices.EnterpriseLibrary.Logging;
using Microsoft.Practices.EnterpriseLibrary.Logging.Formatters;
using Microsoft.Practices.EnterpriseLibrary.Logging.TraceListeners;
```

To configure the application

1. Add the following method to **EntryPoint.cs:**

```
private static LoggingConfiguration BuildProgrammaticConfig()
{
     // Formatter
     TextFormatter formatter = new TextFormatter(
         @"Timestamp:{timestamp(local)}{newline}Message:
         {message}{newline}Category: {category}{newline}Priority:
         {priority}{newline}EventId: {eventid}{newline}ActivityId:
         {property(ActivityId)}{newline}Severity:
         {severity}{newline}Title:{title}{newline}");

     // Trace Listeners
     var eventLog = new EventLog("Application", ".", "EnoughPI");
     var eventLogTraceListener = new FormattedEventLogTraceListener(
        eventLog,
        formatter
      );

     // Build Configuration
     var config = new LoggingConfiguration();
     config.AddLogSource(
        Category.General,
        SourceLevels.All,
        true).AddTraceListener(eventLogTraceListener);
     return config;
}
```

This **LoggingConfiguration** consists of a collection of **LogSources.** Each **LogSource** specifies a name for the source, the **SourceLevel** of logs to include, and a Boolean indicating whether or not to enable auto-flush. Adding a **TraceListener** to the **LogSource** enables that listener to receive logs from the **LogSource** and write them to the specified destination with the selected format. For this application, the **EventLog** source name is set to "EnoughPI." This is the name you will see for log entries when they appear in the Windows Event Log. The **LoggingConfiguration** specifies that these logs will be listed under the "General" category and that all **SourceLevels** should be logged. Finally, a **LogSource** is added to the configuration and the **EventLogTraceListener** you just created is set to listen to that **LogSource**.

2. Change the **Logger** to use this configuration and to dispose it after the application runs in the **Main** method:

```
static void Main()
{
    Application.ThreadException +=
        new System.Threading.ThreadExceptionEventHandler(
          Application_ThreadException);
```

```
    Logger.SetLogWriter(new LogWriter(BuildProgrammaticConfig()));
    Form entryForm = new MainForm();
    Application.EnableVisualStyles();
    Application.Run(entryForm);

    // shut down the logger to flush all buffers
    Logger.Reset();
}
```

The **Logger** façade is a singleton, so setting it in the **EntryPoint** enables the same Log Writer instance to be used in all other classes in this project that require logging (for example, The **Calculator** class) without configuring it more than once.

Note: Using the **Logger** façade is the simplest use of the Logging Application Block, especially when having only one instance of the **LogWriter** class, which is the most common case. Nevertheless, in applications that use an Inversion of Control (IoC) Container, you might consider registering the instance of **LogWriter** directly, so that it can be injected in dependent objects as opposed to those dependent objects using the static façade.

3. Open the **Calc/Calculator.cs** file and add the following using statement.

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

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
     Logger.Write(string.Format("Calculated PI to {0} digits",
          args.Digits),
          Category.General,
          Priority.Normal,
          100);

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

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). This is not required but it is useful for consistency throughout your entire application.

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
  Logger.Write(
      string.Format("Calculating next 9 digits from {0}",
        args.StartingAt),
      Category.General,
      Priority.Low,
      100);

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

  if (args.Cancel == true)
  {
    // TODO: Log cancellation
    Logger.Write("Calculation cancelled by user!",
        Category.General, Priority.High, 100);
  }
}
```

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))
  {
    Logger.Write(args.Exception,
      Category.General,
      Priority.High,
      100);
  }

  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.

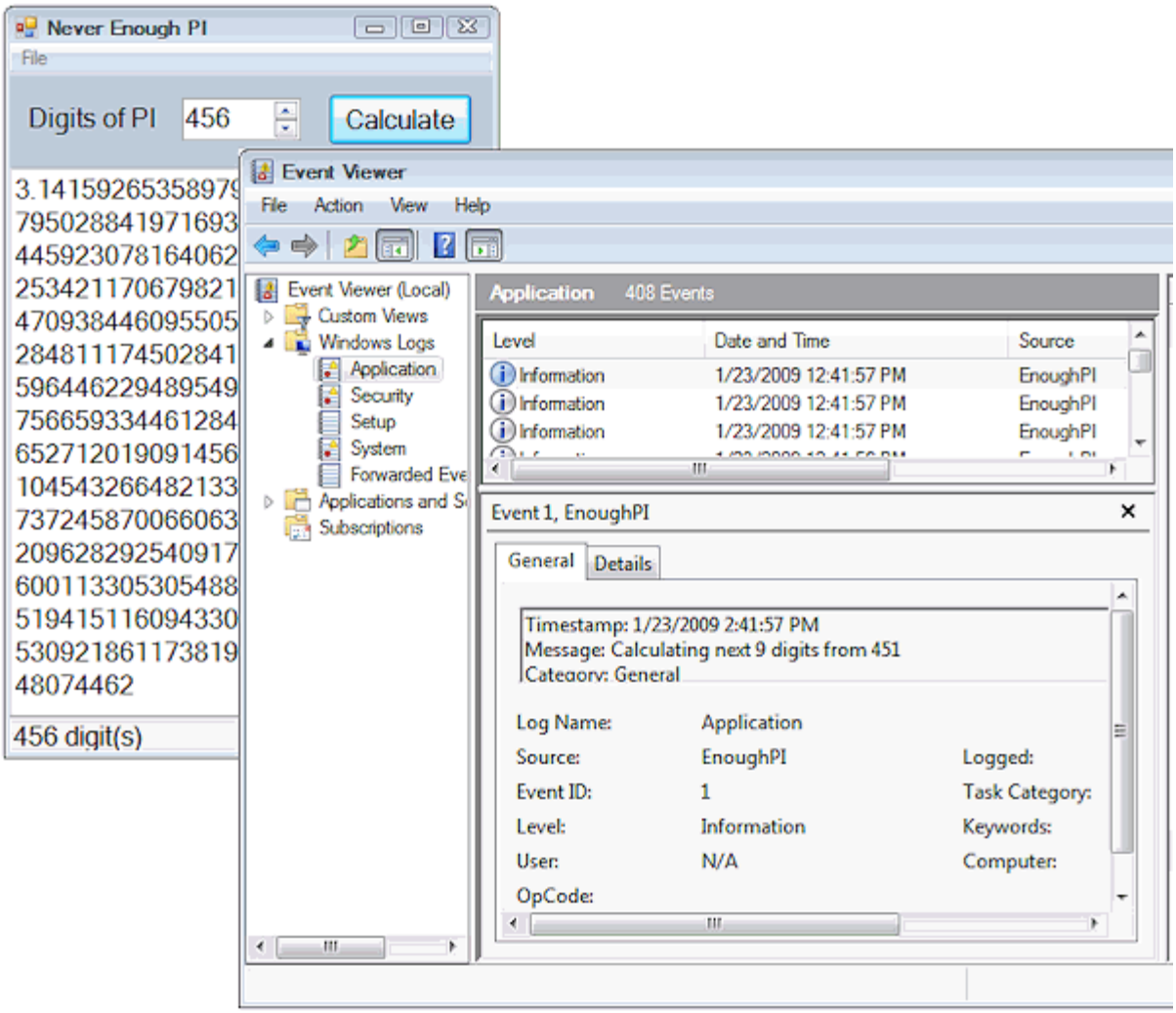
You would normally use the Enterprise Library Exception Handling Application Block to create a consistent strategy for processing exceptions.

To run the application

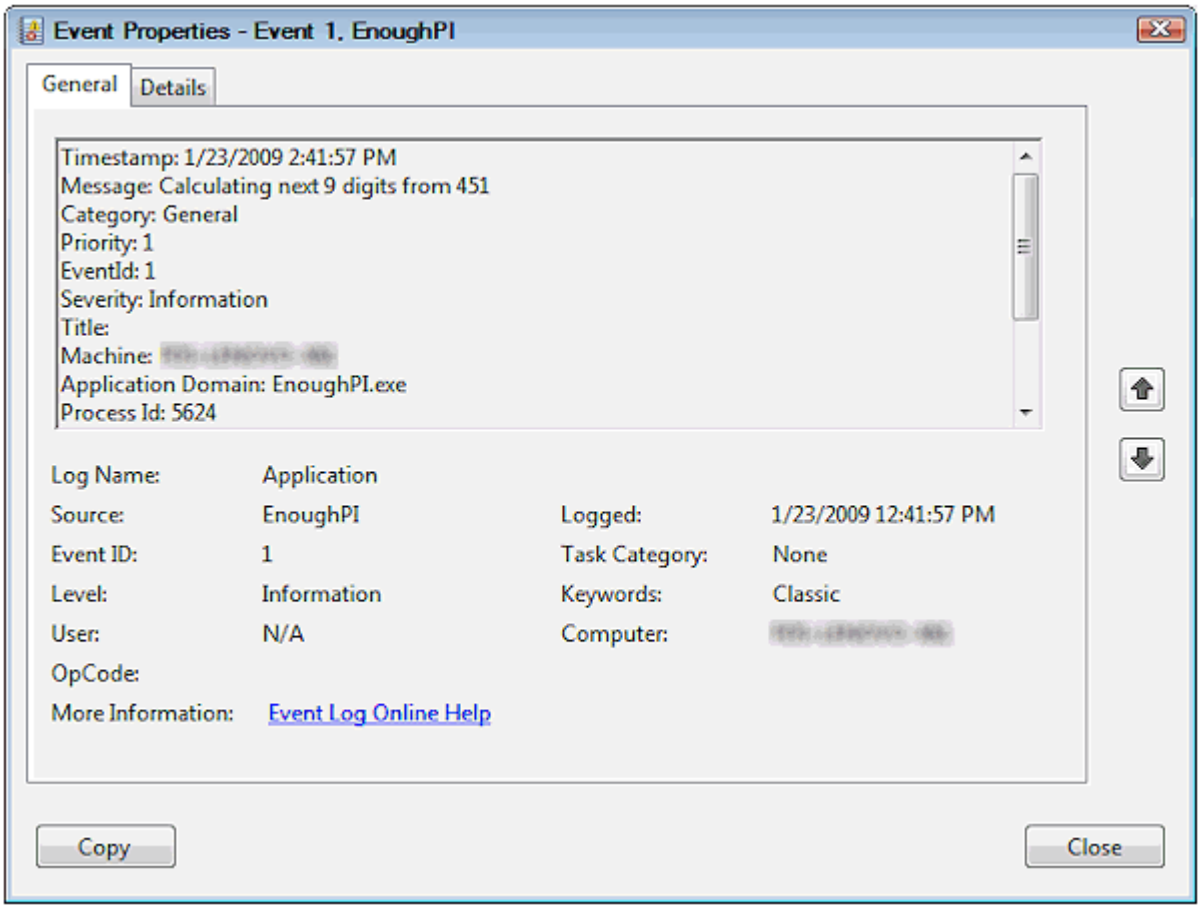
1. Select the **Debug | Start Without Debugging** menu command to run the application. Enter your desired precision and click the **Calculate** button. As mentioned above, the first time you run the application, you'll need to have administrative privileges to get the event source registered.

   > **Note:** Sources used in the event log trace listener 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.

2. Run the event viewer. From the **Control Panel** in Windows, select **Administrative Tools** and then select the **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 screenshot 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 bookend 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 **TraceManager** instance that you will use to trace execution. Initialize the **TraceManager** in the **Calculator** constructor using the **Logger** façade's **LogWriter**:

```
public Calculator()
{
    traceMgr = new TraceManager(Logger.Writer);
}
```

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. Update the **BuildProgrammaticConfig** method in **EntryPoint.cs** to enable tracing.

```
private static LoggingConfiguration BuildProgrammaticConfig()
{
    // Formatter
    TextFormatter formatter = new TextFormatter("Timestamp:
       {timestamp(local)}{newline}Message: {message}{newline}Category:
       {category}{newline}Priority: {priority}{newline}EventId:
       {eventid}{newline}ActivityId:
       {property(ActivityId)}{newline}Severity:
       {severity}{newline}Title:{title}{newline}");

    // Trace Listeners
    var eventLog = new EventLog("Application", ".", "EnoughPI");
    var eventLogTraceListener = new
        FormattedEventLogTraceListener(eventLog, formatter);

    // Build Configuration
    var config = new LoggingConfiguration();
    config.AddLogSource(Category.General, SourceLevels.All,
        true).AddTraceListener(eventLogTraceListener);
    config.IsTracingEnabled = true;
    return config;
}
```

5. Add a new trace listener that will write its output to a disk file and create an event source that utilizes this listener.

```
private static LoggingConfiguration BuildProgrammaticConfig()
{
    // Formatter
    TextFormatter formatter = new TextFormatter("Timestamp:
       {timestamp(local)}{newline}Message: {message}{newline}Category:
       {category}{newline}Priority: {priority}{newline}EventId:
       {eventid}{newline}ActivityId:
       {property(ActivityId)}{newline}Severity:
       {severity}{newline}Title:{title}{newline}");

    // Trace Listeners
    var eventLog = new EventLog("Application", ".", "EnoughPI");
    var eventLogTraceListener = new
       FormattedEventLogTraceListener(eventLog, formatter);
    var flatFileTraceListener = new FlatFileTraceListener(
        @"C:\Temp\trace.log",
        "----------------------------------------",
        "----------------------------------------",
        formatter);

    // Build Configuration
    var config = new LoggingConfiguration();
```

```
    config.AddLogSource(Category.General, SourceLevels.All,
       true).AddTraceListener(eventLogTraceListener);
    config.AddLogSource(Category.Trace, SourceLevels.ActivityTracing,
       true).AddTraceListener(flatFileTraceListener);


     config.IsTracingEnabled = true;

      return config;
}
```

This Flat File Trace Listener will use the same Text Formatter utilized in the Event Log. The event source added is called "Trace", utilizing the constant Category name Trace (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.

To run the application with tracing

1. Select the **Debug | Start Without Debugging** menu command to run the application. Enter your desired precision and click the **Calculate** button.
2. You may view the elapsed time for the trace in the trace.log file, which you will find in the C:\Temp 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 an Asynchronous Trace Listener

In this lab, you will build an Asynchronous Trace Listener Wrapper to write log entries asynchronously to a disk file. Using the asynchronous wrapper changes the perceived time that it takes to log an entry. Control returns to the application faster, but the block still needs to write the log entry to its destination. You will then add this new Trace Listener to the EnoughPI application to monitor the log entries in real time.

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

To monitor how long the log entries take

1. Comment out or remove the **Event Log Trace Listener** from the **BuildProgrammaticConfig** method in **EntryPoint.cs** so you are only keeping track of the time it takes to log using your **Flat File Trace Listener.**

```
private static LoggingConfiguration BuildProgrammaticConfig()
    {
        // Formatter
        TextFormatter formatter = new TextFormatter(@"Timestamp:
{timestamp(local)}{newline}Message: {message}{newline}Category:
{category}{newline}Priority: {priority}{newline}EventId:
{eventid}{newline}ActivityId: {property(ActivityId)}{newline}Severity:
{severity}{newline}Title:{title}{newline}");
        var xmlFormatterAttributes = new NameValueCollection();
        xmlFormatterAttributes["prefix"] = "x";
        xmlFormatterAttributes["namespace"] = "EnoughPI/2.0";
        EnoughPI.Logging.Formatters.XmlFormatter xmlFormatter =
          new EnoughPI.Logging.Formatters.XmlFormatter(
            xmlFormatterAttributes);

        // Trace Listeners
        var eventLog = new EventLog("Application", ".", "EnoughPI");
        var eventLogTraceListener = new
FormattedEventLogTraceListener(eventLog, formatter);
        var flatFileTraceListener =
          new FlatFileTraceListener(
            @"C:\Temp\trace.log",
            "----------------------------------------",
            "----------------------------------------",
            formatter);

        // Build Configuration
        var config = new LoggingConfiguration();
```

```
        config.AddLogSource(Category.General, SourceLevels.All,
true).AddTraceListener(eventLogTraceListener);
        config.AddLogSource(
           Category.Trace,
           SourceLevels.ActivityTracing,
           true).AddTraceListener(flatFileTraceListener);

        return config;
    }
```

2. Select the **Debug | Start Without Debugging** menu command to run the application. Enter a precision of at least 300 (this will make the time improvements more apparent) and click the **Calculate** button. The end of the Tracing logs in **C:\Temp\trace.log** will tell how long it took to calculate pi.

```
----------------------------------------
Timestamp: 7/22/2013 8:29:13 AM
Message: End Trace: Activity '67ba73cf-502c-4c3d-bc04-c2ea11c7e88f' in
method 'EnoughPI.Calc.Calculator.Calculate' at 1194567702026 ticks
(elapsed time: 3.554 seconds)
Category: Trace
Priority: 5
EventId: 1
ActivityId: 67ba73cf-502c-4c3d-bc04-c2ea11c7e88f
Severity: Stop
Title:TracerExit

----------------------------------------
----------------------------------------
Timestamp: 7/22/2013 8:29:13 AM
Message: Calculated PI to 300 digits
Category: General
Priority: 2
EventId: 100
ActivityId: 00000000-0000-0000-0000-000000000000
Severity: Information
Title:

----------------------------------------
```

To use a trace listener asynchronously

1. Use the **AddAsynchronousTraceListener** method in the **BuildProgrammaticConfig** method in **EntryPoint.cs** to add the flatFileTraceListener to your configuration.

```
private static LoggingConfiguration BuildProgrammaticConfig()
{
    // Formatter
    TextFormatter formatter = new TextFormatter("Timestamp:
        {timestamp(local)}{newline}Message:
```

```
        {message}{newline}Category: {category}{newline}Priority:
        {priority}{newline}EventId: {eventid}{newline}ActivityId:
        {property(ActivityId)}{newline}Severity:
        {severity}{newline}Title:{title}{newline}");

    // Trace Listeners
    var flatFileTraceListener = new
      FlatFileTraceListener(@"C:\Temp\trace.log", "-------------------
      ---------------------", "-------------------------------------
      -", formatter);

    // Build Configuration
    var config = new LoggingConfiguration();
    config.AddLogSource(Category.Trace, SourceLevels.ActivityTracing,
true).AddAsynchronousTraceListener(flatFileTraceListener);
    config.IsTracingEnabled = true;


    return config;
```

Wrapping the existing **FlatFileTraceListener** allows you to use that **Trace Listener** to log messages asynchronously. This will be most useful when writing large volumes of messages to a flat file or database.

2. View the output again. It should be significantly lower now.

```
----------------------------------------
Timestamp: 7/22/2013 8:33:54 AM
Message: End Trace: Activity '00c3f38c-233c-4d46-9958-19df15242634' in
method 'EnoughPI.Calc.Calculator.Calculate' at 1195224522966 ticks
(elapsed time: 0.912 seconds)
Category: Trace
Priority: 5
EventId: 1
ActivityId: 00000000-0000-0000-0000-000000000000
Severity: Stop
Title:TracerExit

----------------------------------------
----------------------------------------
Timestamp: 7/22/2013 8:33:54 AM
Message: Calculated PI to 300 digits
Category: General
Priority: 2
EventId: 100
ActivityId: 00000000-0000-0000-0000-000000000000
Severity: Information
Title:
```

Note: Logging messages asynchronously can lead to messages being lost if the application terminates before the buffer is drained. Disposing the LogWriter when shutting down the application attempts to flush all asynchronous buffers.

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

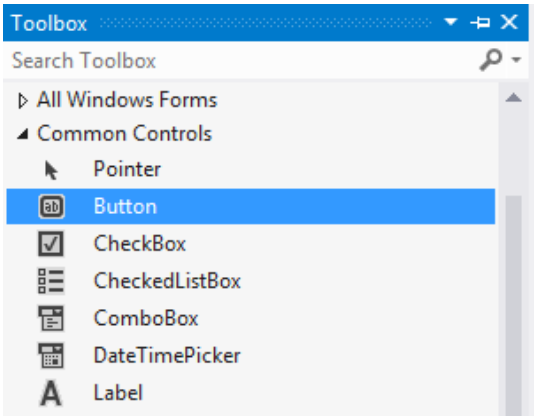
Lab 3: Reconfigure Logging at Runtime

In this lab, you will dynamically change the logging configuration while the application is running. This allows you to make temporary changes to your logging configuration to collect additional information without stopping and starting your application.

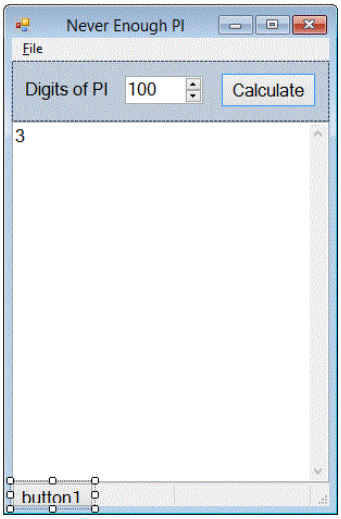
To begin this exercise, open the EnoughPI.sln file located in the ex03\begin folder.

To create a button in the application to update the configuration

1. Click on **MainForm.cs** and Select **View|Designer** to edit the layout of the form.
2. Select the **Toolbox** panel at the left of the screen and view the **Common Controls** section.



3. Select the **Button** control and drag it to the form. Position it in the bottom left corner.



4. Right-click the **Button** you just added and select the **Properties** menu item.
5. Change the **Text** property to “Update.”

6. Select the lightning bolt **Events** icon to edit the events for the **Button.** Under the **Action** category, double-click the blank box next to **Click** and it will fill in "button1_Click." This generates a method to be executed when the button is clicked. You will update this method in the next task.

To update the logging configuration

1. Add the following using statements to **MainForm.cs**.

```
using EnoughPI.Logging;
using Microsoft.Practices.EnterpriseLibrary.Logging;
using Microsoft.Practices.EnterpriseLibrary.Logging.TraceListeners;
using Microsoft.Practices.EnterpriseLibrary.Logging.Formatters;
```

2. Create a member variable to track how many times the configuration has been updated and initialize it to zero.

```
private int numUpdates = 0;
```

3. Create a method called **UpdateConfiguration** in **MainForm.cs** to change the destination of the log message.

```
public void UpdateConfiguration()
{
    numUpdates++;
    Logger.Writer.Configure(config =>
    {
        TextFormatter formatter = new TextFormatter(@"Timestamp:
           {timestamp(local)}{newline}Message:
           {message}{newline}Category: {category}{newline}Priority:
           {priority}{newline}EventId: {eventid}{newline}ActivityId:
           {property(ActivityId)}{newline}Severity:
           {severity}{newline}Title:{title}{newline}");
        var newDest = new FlatFileTraceListener(@"C:\Temp\trace-" +
           numUpdates + @".log", "----------------------------------
           ", "----------------------------------------", formatter);
        config.SpecialSources.AllEvents.Listeners.Clear();
        config.SpecialSources.AllEvents.AddTraceListener(newDest);

    });
}
```

This function reconfigures your **Logger**'s **LogWriter** to write to a new destination: a new **FlatFileTraceListener** whose name reflects the number of times the configuration has been updated. First it clears the Listeners for the **Trace** category, then adds the new **FlatFileTraceListener.**

4. Call your **UpdateConfiguration** method from the **button1_Click** method in **MainForm.cs**.

```
private void button1_Click(object sender, EventArgs e)
```

```
{
    UpdateConfiguration();
}
```

To verify reconfiguration at run time is successful

1. Select the **Debug | Start Without Debugging** menu command to run the application. Enter your desired precision (something greater than 300 would be best) and click the **Calculate** button.
2. You may view the trace files in the C:\Temp folder. There should be multiple trace files, detailing the number of times the update button was clicked the file starts at (for example, trace-1.log.)
3. Close the application and Visual Studio.

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

Lab 4: 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 ex04\begin folder.

To create a custom Trace Listener

1. 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.TraceListeners;
```

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

```
[ConfigurationElementType(typeof(CustomTraceListenerData))]
public class ConsoleTraceListener : CustomTraceListener
{
  public ConsoleTraceListener()
    : base()
  {
  }
  public ConsoleTraceListener(string del)
  {
     this.Attributes["delimiter"] = del;
  }

  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);
  }
}
```

Note: 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.

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

To use a custom Trace Listener

1. In **EntryPoint.cs** add your **CustomTraceListener** to your logging configuration in the **BuildProgrammaticConfig** method.

```
private static LoggingConfiguration BuildProgrammaticConfig()
{
     // Formatter
     TextFormatter formatter = new TextFormatter(@"Timestamp:
     {timestamp(local)}{newline}Message: {message}{newline}Category:
     {category}{newline}Priority: {priority}{newline}EventId:
     {eventid}{newline}ActivityId:
     {property(ActivityId)}{newline}Severity:
     {severity}{newline}Title:{title}{newline}");

     // Trace Listeners
     var eventLog = new EventLog("Application", ".", "EnoughPI");
     var eventLogTraceListener = new
         FormattedEventLogTraceListener(eventLog, formatter);
     var flatFileTraceListener = new
           FlatFileTraceListener(
              @"C:\Temp\trace.log",
              "----------------------------------------",
              "----------------------------------------",
```

```
            formatter);

    var customTraceListener =
       new EnoughPI.Logging.TraceListeners.ConsoleTraceListener(
           "----------------------");

    // Build Configuration
    var config = new LoggingConfiguration();
    config.AddLogSource(
        Category.General,
        SourceLevels.All,
        true).AddTraceListener(eventLogTraceListener);
    config.AddLogSource(
        Category.Trace,
        SourceLevels.ActivityTracing,
        true).AddTraceListener(flatFileTraceListener);

    config.LogSources[Category.General].AddTraceListener(
        customTraceListener);
    config.IsTracingEnabled = true;

    return config;
}
```

Because the LogSource named Category.General already exists, you can add another trace listener to it by referencing the LogSource by name, as above.

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

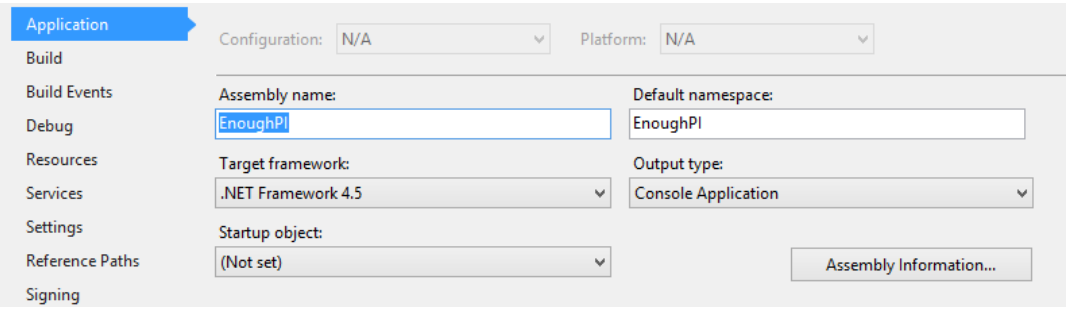
2. Set the **Formatter** to the **Text Formatter** you created earlier.

```
private static LoggingConfiguration BuildProgrammaticConfig()
{
            // Formatter
            TextFormatter formatter = new TextFormatter(@"Timestamp:
{timestamp(local)}{newline}Message: {message}{newline}Category:
{category}{newline}Priority: {priority}{newline}EventId:
{eventid}{newline}ActivityId: {property(ActivityId)}{newline}Severity:
{severity}{newline}Title:{title}{newline}");

    // Trace Listeners
    var eventLog = new EventLog("Application", ".", "EnoughPI");
    var eventLogTraceListener =
        new FormattedEventLogTraceListener(eventLog, formatter);
    var flatFileTraceListener =
        new FlatFileTraceListener(
           @"C:\Temp\trace.log",
           "----------------------------------------",
```

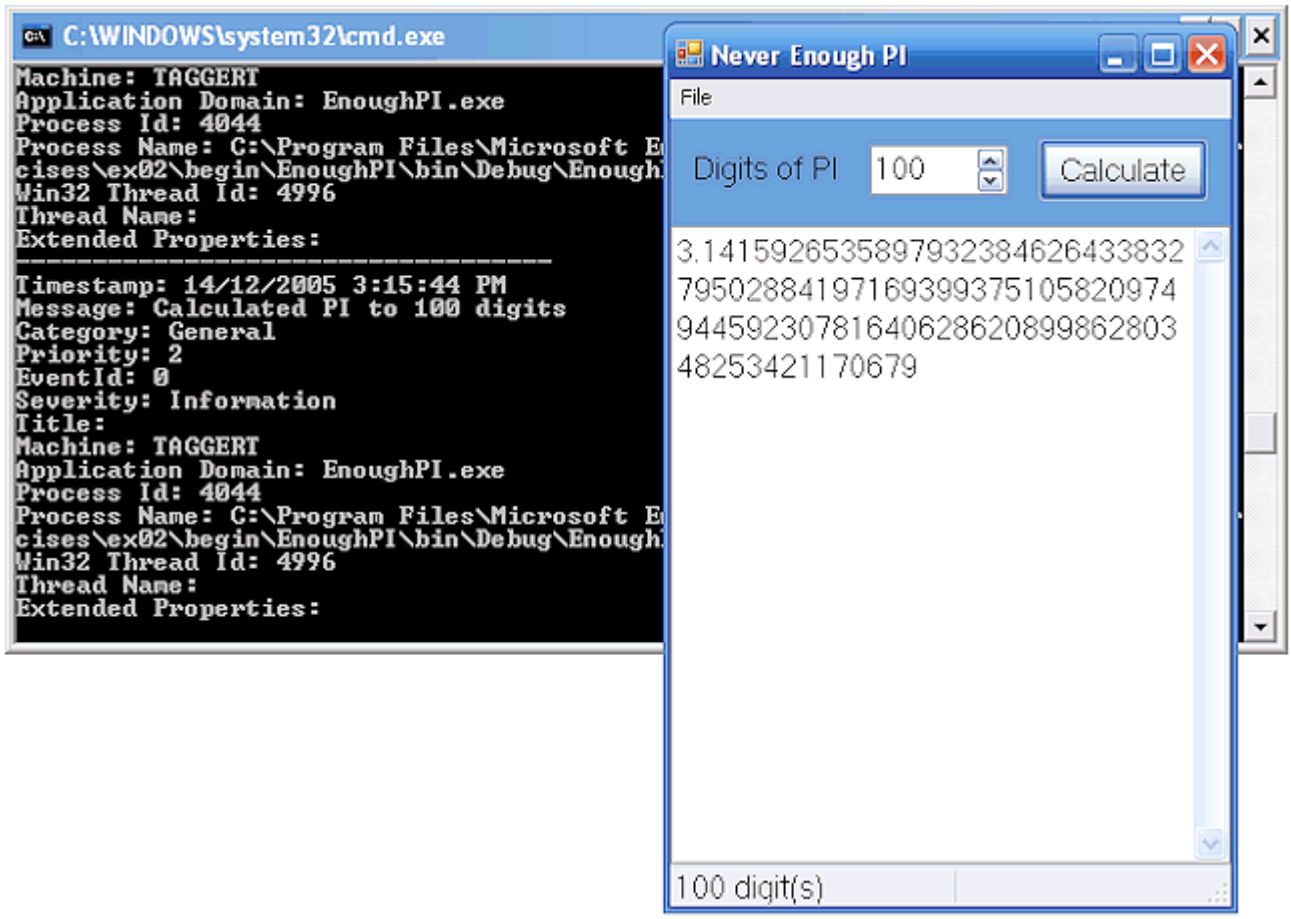
```
            "----------------------------------------",
            formatter);
    var customTraceListener =
        new EnoughPI.Logging.TraceListeners.ConsoleTraceListener(
            "----------------------");
    customTraceListener.Formatter = formatter;

   // Build Configuration
   var config = new LoggingConfiguration();
   config.AddLogSource(
       Category.General,
       SourceLevels.All,
       true).AddTraceListener(eventLogTraceListener);
   config.AddLogSource(
       Category.Trace,
       SourceLevels.ActivityTracing,
       true).AddTraceListener(flatFileTraceListener);
   config.LogSources[Category.General].AddTraceListener(
           customTraceListener);


   config.IsTracingEnabled = true;

   return config;
}
```

To view the Trace Listener output

1. Select the **EnoughPI** project. Select the **Project | EnoughPI Properties…** menu command, select the **Application** tab, and set **Output type** to **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 ex04\end folder.

Lab 5: 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 ex05\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 XmlFormatter(string prefix, string ns)
  {
    this.Attributes = new NameValueCollection();
    this.Attributes["prefix"] = prefix;
    this.Attributes["namespace"] = ns;
  }

  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("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.WriteEndElement();
            w.WriteEndDocument();

            return s.ToString();
        }
    }
}
```

The log entry will be formatted as XML. The built-in **XmlLogFormatter** is useful, but not easily human readable. By creating a custom formatter, you ensure that only the information you care about is included and the information is formatted in a way that makes sense for your purposes. This is accomplished by overriding the **Format** function of the **LogFormatter** parent class. Here, you include the Priority, Timestamp, Message, Event Id, Severity, Machine, App Domain, Process Id, Process Name, and Thread Id. Also, you set the **XmlTextWriter**'s **Formatting** attribute to "Indented," making the logs much easier to read.

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

To use a custom log formatter

1. In the **BuildProgrammaticConfig** method in **EntryPoint.cs** add an **XmlFormatter.**

```
private static LoggingConfiguration BuildProgrammaticConfig()
{
    // Formatter
    TextFormatter formatter = new TextFormatter("Timestamp:
       {timestamp(local)}{newline}Message: {message}{newline}Category:
       {category}{newline}Priority: {priority}{newline}EventId:
       {eventid}{newline}ActivityId:
       {property(ActivityId)}{newline}Severity:
       {severity}{newline}Title:{title}{newline}");
    var xmlFormatter = new
       EnoughPI.Logging.Formatters.XmlFormatter("x", "EnoughPI/2.0");
```

```
    // Trace Listeners
    var eventLog = new EventLog("Application", ".", "EnoughPI");
    var eventLogTraceListener = new
       FormattedEventLogTraceListener(eventLog, formatter);
    var flatFileTraceListener = new
       FlatFileTraceListener(
           @"C:\Temp\trace.log",
           "----------------------------------------",
           "----------------------------------------",
formatter);
    var customTraceListener =  new
        EnoughPI.Logging.TraceListeners.ConsoleTraceListener(
           "-----------------------");

    customTraceListener.Formatter = xmlFormatter;

    // Build Configuration
    var config = new LoggingConfiguration();
    config.AddLogSource(Category.General, SourceLevels.All,
        true).AddTraceListener(eventLogTraceListener);
    config.AddLogSource(Category.Trace,
        SourceLevels.ActivityTracing,
        true).AddTraceListener(flatFileTraceListener);


config.LogSources[Category.General].AddTraceListener(customTraceListen
er);
    config.IsTracingEnabled = true;


    return config;
}
```

The **XmlFormatter** constructor expects a collection of attributes, specifically a **prefix** and **namespace**. Set **prefix** as “x” and **namespace** as “EnoughPI/2.0.” Set the **Custom Trace Listener** you created in the previous lab to use this formatter.

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 ex05\end folder.

More Information

For more information about the Logging Application Block, see the documentation in the Enterprise Library 6 Developer's Guide and the Enterprise Library 6 Reference documentation.

patterns & practices
proven practices for predictable results

Copyright