Instructor Notes:

Instructor Notes:

Explain the lesson coverage

Lesson Objective

- In this lesson, you will learn:
  - Understand Trace Source and Logging

Instructor Notes:

What is TraceSource?

- Provides a set of methods and properties that enable applications to trace the execution of code and assThe
- TraceSource class is identified by the name of a source, typically the name of the application. The trace messages coming from a particular component can be initiated by a particular trace source, allowing all messages coming from that component to be easily identified.
- TraceSource defines tracing methods but does not actually provide any specific mechanism for generating and storing tracing data. The tracing data is produced by trace listeners, which are plug-ins that can be loaded by trace sourceslociated trace messages with their source.

```
using System;
using System.Diagnostics;
using System.Threading;

namespace TraceSourceApp
{
    class Program
    {
        private static TraceSource mySource =
            new TraceSource("TraceSourceApp");
        static void Main(string[] args)
        {
            Activity1();
            mySource.Close();
            return;
        }
        static void Activity1()
        {
            mySource.TraceEvent(TraceEventType.Error, 1,
```

```
                "Error message.");
            mySource.TraceEvent(TraceEventType.Warning, 2,
                "Warning message.");
        }
    }
}
```

Instru

Instructor Notes:

What is TraceListeners?

- When using Trace, Debug and TraceSource, you must have a mechanism for collecting and recording the messages that are sent. Trace messages are received by listeners. The purpose of a listener is to collect, store, and route tracing messages. Listeners direct the tracing output to an appropriate target, such as a log, window, or text file.
- Listeners are available to the Debug, Trace, and TraceSource classes, each of which can send its output to a variety of listener objects. The following are the commonly used predefined listeners:
- A TextWriterTraceListener redirects output to an instance of the TextWriter class or to anything that is a Stream class. It can also write to the console or to a file, because these are Stream classes

```
using System;
using System.Diagnostics;
using System.Threading;

namespace TraceSourceApp
{
    class Program
    {
        private static TraceSource mySource =
            new TraceSource("TraceSourceApp");
        static void Main(string[] args)
        {
            Activity1();

            // Change the event type for which tracing occurs.
            // The console trace listener must be specified
            // in the configuration file. First, save the original
            // settings from the configuration file.
            EventTypeFilter configFilter =
```

```
            (EventTypeFilter)mySource.Listeners["console"].Filter;

        // Then create a new event type filter that ensures
        // critical messages will be written.
        mySource.Listeners["console"].Filter =
            new EventTypeFilter(SourceLevels.Critical);
        Activity2();

        // Allow the trace source to send messages to listeners
        // for all event types. This statement will override
        // any settings in the configuration file.
        mySource.Switch.Level = SourceLevels.All;

        // Restore the original filter settings.
        mySource.Listeners["console"].Filter = configFilter;
        Activity3();
        mySource.Close();
        return;
    }
    static void Activity1()
    {
        mySource.TraceEvent(TraceEventType.Error, 1,
            "Error message.");
        mySource.TraceEvent(TraceEventType.Warning, 2,
            "Warning message.");
    }
    static void Activity2()
    {
        mySource.TraceEvent(TraceEventType.Critical, 3,
            "Critical message.");
        mySource.TraceInformation("Informational message.");
    }
    static void Activity3()
    {
        mySource.TraceEvent(TraceEventType.Error, 4,
            "Error message.");
        mySource.TraceInformation("Informational message.");
    }
}
```

```
        }
```

Instru

Instructor Notes:

What is TraceListeners?

- An EventLogTraceListener redirects output to an event log.
- A DefaultTraceListener emits Write and WriteLine messages to the OutputDebugString and to the Debugger.Log method. In Visual Studio, this causes the debugging messages to appear in the Output window. Fail and failed Assert messages also emit to the OutputDebugString Windows API and the Debugger.Log method, and also cause a message box to be displayed. This behavior is the default behavior for Debug and Trace messages, because DefaultTraceListener is automatically included in every Listeners collection and is the only listener automatically included.

```
using System;
using System.Diagnostics;
using System.Threading;

namespace TraceSourceApp
{
    class Program
    {
        private static TraceSource mySource =
            new TraceSource("TraceSourceApp");
        static void Main(string[] args)
        {
            Activity1();

            // Change the event type for which tracing occurs.
            // The console trace listener must be specified
            // in the configuration file. First, save the original
            // settings from the configuration file.
            EventTypeFilter configFilter =
```

```
            (EventTypeFilter)mySource.Listeners["console"].Filter;
```

Instru

```
            // Then create a new event type filter that ensures
            // critical messages will be written.
            mySource.Listeners["console"].Filter =
                new EventTypeFilter(SourceLevels.Critical);
            Activity2();

            // Allow the trace source to send messages to listeners
            // for all event types. This statement will override
            // any settings in the configuration file.
            mySource.Switch.Level = SourceLevels.All;

            // Restore the original filter settings.
            mySource.Listeners["console"].Filter = configFilter;
            Activity3();
            mySource.Close();
            return;
        }
        static void Activity1()
        {
            mySource.TraceEvent(TraceEventType.Error, 1,
                "Error message.");
            mySource.TraceEvent(TraceEventType.Warning, 2,
                "Warning message.");
        }
        static void Activity2()
        {
            mySource.TraceEvent(TraceEventType.Critical, 3,
                "Critical message.");
            mySource.TraceInformation("Informational message.");
        }
        static void Activity3()
        {
            mySource.TraceEvent(TraceEventType.Error, 4,
                "Error message.");
            mySource.TraceInformation("Informational message.");
        }
    }
```

```
        }
```

Instru

Instructor Notes:

What is TraceListeners?

- A ConsoleTraceListener directs tracing or debugging output to either the standard output or the standard error stream.
- A DelimitedListTraceListener directs tracing or debugging output to a text writer, such as a stream writer, or to a stream, such as a file stream. The trace output is in a delimited text format that uses the delimiter specified by the Delimiter property.
- An XmlWriterTraceListener directs tracing or debugging output as XML-encoded data to a TextWriter or to a Stream, such as a FileStream.

Instructor Notes:

WPF Tracing

- Visual Studio can receive debug trace information from WPF applications and display that information in the **Output** window. To display debug trace information, WPF tracing must be enabled.

To enable or customize WPF trace information

On the **Tools** menu, select **Options**.

In the **Options** dialog box, in the box on the left, open the **Debugging** node.

Under **Debugging**, click **Output Window**.

Under **General Output Settings**, select **All debug output**.

In the box on the right, look for **WPF Trace Settings**.

Open the **WPF Trace Settings** node.

Under **WPF Trace Settings**, click the category of settings that you want to enable (for example, **Data Binding**).

A drop-down list control appears in the Settings column next to **Data Binding** or whatever category you clicked.

Click the drop-down list and select the type of trace information that you want to see: **All**, **Critical**, **Error**, **Warning**, **Information**, **Verbose**, or **ActivityTracing**.

Critical enables tracing of Critical events only.

Error enables tracing of Critical and Error events.

Warning enables tracing of Critical, Error, and Warning events.

Information enables tracing of Critical, Error, Warning, and Information events.

Verbose enables tracing of Critical, Error, Warning, Information, and Verbose

events.

Instru

ActivityTracing enables tracing of Stop, Start, Suspend, Transfer, and Resume events.

For more information about what these levels of trace information mean, see SourceLevels.

Click **OK**.

To disable WPF trace information

On the **Tools** menu, select **Options**.

In the **Options** dialog box, in the box on the left, open the **Debugging** node.

Under **Debugging**, click **Output Window**.

In the box on the right, look for **WPF Trace Settings**.

Open the **WPF Trace Settings** node.

Under **WPF Trace Settings**, click the category of settings that you want to enable (for example, **Data Binding**).

A drop-down list control appears in the Settings column next to **Data Binding** or whatever category you clicked.

Click the drop-down list and select **Off**.

Click **OK**.

Instructor Notes:

Overview of Logging

- Logging is writing the state of a program at various stages of its execution to some repository such as a log file
- By logging, explanatory statements can be sent to text file, console, or any other repository
- Using logging, a reliable monitoring and debugging solution can be achieved

Logging is used due to the following reasons:

It can be used for debugging.

It is cost effective than including some debug flag.

There is no need to recompile the program to enable debugging.

It does not leave your code messy.

Priority levels can be set.

Log statements can be appended to various destinations such as file, console, socket, database, and so on.

Logs are needed to quickly target an issue that occurred in service.

Instructor Notes:

What is Log4Net?

- Log4Net is an open source logging API for .NET from Apache Software Foundation
- log4net is a tool to help the programmer output log statements to a variety of output targets
- With log4net it is possible to enable logging at runtime without modifying the application binary

log4net provides many advantages over other logging systems, which makes it a perfect choice for use in any type of application, from a simple single-user application to a complex multiple-threaded distributed application using remoting

Instructor Notes:

Features of Log4Net

- Support for multiple frameworks
  - .NET Framework 1.1, 2.0, Mono
- Output to multiple logging targets
  - Database, Terminal window, ASP trace context, Applications window Console, Window event log, File System etc
- Hierarchical logging architecture
- Dynamic XML Configuration

Hierarchical logging is an ideal fit with component based development. Each component has its own of logger. When individually tested, the properties of these loggers may be set as the developer requires. When combined with other components, the loggers inherit the properties determined by the integrator of the components. One can selectively elevate logging priorities on one component without affecting the other components. This is useful when you need a detailed trace from just a single component without crowding the trace file with messages from other components. All this can be done through configuration files; no code changes are required.

log4net is configured using an XML configuration file. The configuration information can be embedded within other XML configuration files (such as the application's .config file) or in a separate file. The configuration is easily readable and updateable while retaining the flexibility to express all configurations. Alternatively log4net can be configured programmatically.

Instructor Notes:

Components of Log4Net

- Log4net is built using the layered approach, with the following components inside of the framework
  - Logger, Appender, & Layout
- Users can extend these basic classes to create their own loggers, appenders, and layouts

The Logger is the main component with which your application interacts. It is also the component that generates the log messages.

Generating a log message is different than actually showing the final output. The output is showed by the Layout component.

The logger provides you with different methods to log any message. You can create multiple loggers inside of your application.

Any good logging framework should be able to generate output for multiple destinations, such as outputting the trace statements to the console or serializing it into a log file. log4net is a perfect match for this requirement. It uses a component called *Appender* to define this output medium. As the name suggests, these components append themselves to the Logger component and relay the output to an output stream. You can append multiple appenders to a single logger.

The Layout component is used to display the final formatted output to the user. The output can be shown in multiple formats, depending upon the layout we are using. It can be linear or an XML file. The layout component works with an appender. There is a list of different layouts in the API documentation. You cannot use multiple layouts with an appender. To create your own layout, you need to inherit the log4net.Layout.LayoutSkeleton class, which implements the ILayout interface.

Instructor Notes:

Demo

- Demo on Log4Net