# Design guide for TraceListener extensions

The two main issues when creating new TraceListener-derived classes are configuration and method overrides. An understanding of how a TraceListener is created by System.Diagnostics, how the methods are called, and what features are provided by the base TraceListener class will inform the design of new classes

## Configuration issues

### TraceListener via configuration

When creating listeners from the config file, the initializeData attribute is used in the constructor if possible.

If initializeData is empty, the empty constructor is used.

If initializeData is not empty, then a constructor with a single argument is looked for:

First, if there is a constructor with a single string then it is used. The configuration loader also does special munging for when creating Framework listeners TextWriterTraceListener, DelimitedListTraceListener and XmlWriterTraceListener (but not derived classes), where if the string is a relative path it is converted to an absolute path relative to the location of the config file. Note that this special munging only occurs when creating from the config file – if created in code, then relative paths may be used in which case they are relative to the location of the running executable.

Otherwise, all constructors are checked for any that take a single parameter. For all those found an attempt is made to either parse initializeData (if the parameter is an Enum), or convert to the parameter type (e.g. Boolean). The first constructor the data can be parsed/converted to is used. Usually this would be used for constructors that take a single containing single Enum, but could also be another simple type such as Boolean (used in ConsoleTraceListener), number, etc.

After constructing the listener, the configuration loader then sets Name, Attributes and TraceOutputOptions.

### TraceListener via code

The base TraceListener class has two constructors – one empty, and one with a single string parameter for name.

The empty base constructor is used when loading from the configuration file and there is no relation between the constructor with the name string and initializeData. The name is set separately from the Name configuration property; any strings in the initializeData are used for other purposes, e.g. filename.

When creating from code, however, TraceListeners may have an additional overloaded constructor that takes a second string parameter and is used as a convenience for setting the name property.

### Implications for designing TraceListener extensions

Provide an empty constructor if appropriate and a constructor that takes a single attribute appropriate to be put in initializeData. If replacing Framework listeners, the type of initializeData should be consistent, e.g. if writing to a file it should be a filename; for a console listener it should be a Boolean specifying useErrorStream.

Usually the initializeData specifies the location of the trace, e.g. the location of the file, which console stream to use, or which Event Log source to use.

This means for a database-based listener it should probably be the connection string or database location information, for MSMQ it should be the queue name, etc.

If possible, provide default values for other properties, so that additional attributes can be optional. For example, the Delimiter in the DelimitedListTraceListener is optional and has a reasonable default. This is especially important if providing expanded behavior to replace existing Framework listeners – the default should be close to, or a reasonable extension of, the Framework behavior.

e.g. ColoredConsoleTraceListener provides formatting and coloring enhancements compared ConsoleTraceListener, but accepts the same initializeData, defaults to a matching format string, and has a reasonable set of default colors. The format can then be changed by the formatString attribute, and colors changed by errorColor, warningColor, etc.

### Additional configuration attributes

Common Framework attributes are Name, InitializeData and TraceOutputOptions. Additional options found in Framework classes include delimiter in DelimitedListTraceListener and EventProviderTraceListener as well as a few attributes in EventSchemaTraceListener and half a dozen attributes in FileLogTraceListener.

Where attributes provide the same function, they should use the same name as existing attributes. In other cases, attribute names should be consistent between listeners.

As mentioned above, additional attributes should be optional and provide reasonable default values.

## Method overrides

### Default behavior of trace functions

The base TraceListener class has two abstract functions WriteLine(string) and Write(string), with all the other signature variations call through to these two.

If, however, you want to provide alternative formatting or handling based on the different arguments, then you need to override the other virtual methods such as TraceData(), TraceEvent, etc.

The default behavior of TraceData() and TraceEvent() is to check the filter ShouldTrace(), then build one line containing “Source EventTypeString: Id : ( Message | Format | Data0 [[,Data1] …] )”, and then, if the eventCache is set, additional indented lines for each trace option set: ProcessId, LogicalOperationstack, ThreadId, DateTime, Timestamp, Callstack.

The two TraceData() methods each format this independently and then call WriteLine() / Write(), as do the TraceEvent() methods with and without format args (this means the args aren’t formatted until after ShouldTrace() is checked).

Also, the TraceEvent() method without a message calls TraceEvent() with an empty message, and TraceTransfer() calls TraceEvent() with “, relatedActivityId=” appended to the message.

Note also that the TraceInformation() methods of TraceSource call TraceEvent() with TraceEventType.Information and id 0.

### Static methods on Trace

It is recommended that the newer methods on TraceSource be used, as they provide more consistent results, however you may need to interoperate with older code so it is important to know how a listener could be called.

There are three groups of methods on the static Trace class that need to be considered: the Trace…() methods, the Write…() method overloads, and the base Write…() methods.

The TraceError(), TraceWarning() and TraceInformation() static methods on Trace call TraceEvent with the appropriate TraceEventType and id 0, which means they are formatted as above (and eventually forwarded to WriteLine/Write).

The overloads WriteLine/Write(object), WriteLine/Write (object,string) and WriteLine/Write (string,string) all first of all check the filter ShouldTrace() as if the message were being logged at Verbose level (with id 0), then they call ToString() on the object and/or prepend “Category: “, before calling the base WriteLine/Write().

Note that this behavior is inconsistent with the WriteLineIf methods, where if you call WriteLineIf(bool,object) after checking for, say, Error level, the implementation of WriteLine(object) then also checks Verbose level.

The third group is the base WriteLine/Write methods, which can also be called directly, yet they are also called by the default implementations of the Trace…() methods, so they aren’t filtered (like the other Write methods).

Effectively, this means that when using a stream-based trace, WriteLine/Write are written directly to the stream. If using structured output, such as the Event Log (or a database, etc), then “Writing” does not really make sense and these method calls need to be converted to an event. An example of this is the EventLogTraceListener.

### Overriding TraceListener methods

For the limited case where you have a stream-based trace and where you are happy with the format used by the default TraceListener, then you only need to implement the Write() and WriteLine() methods.

Usually, however if you want be able to change the format or support structured trace outputs. For example outputting to a database or XML structure, or when FileLogTraceListener uses a different text format.

This means in most cases you need to override the two TraceData() methods and the two TraceEvent() methods with message arguments. If you want to separately track the relatedActivityId then you also need to override TraceTransfer().

You also need to still implement the Write() and WriteLine(), usually as Verbose events with id=0 (similar to the conditions checked by ShouldTrace() in the base listener), i.e. rather than forwarding the trace methods to write, you should treat the write methods as Verbose traces.

Another option if you are still outputting to a stream (but wanted to handle the Trace…() methods with alternative formatting) is you could still output WriteLine/Write() to the stream, or even provide a convertWriteToEvent attribute to allow the behavior to be configured between directly writing to the stream and treating as a Verbose event.