Documentation

The logging framework is a very lightweight, generic and flexible tool for relaying debug and failure information. A small number of predefined "Loggers" (log message consumers) is provided (e. g. for logging to text files, to the console and to various GUI controls in Windows Forms as well as Windows Presentation Foundation). No setup or initialization whatsoever is required to use the framework.

The following code shows how to use the framework to log string messages to a RichTextBox inside a windows forms application assuming the RichTextBox is named "richTextBox1":

```
Logger.Add(new RichTextBoxLogger(richTextBox1));
// ...
this.Log<INFO>("Hello World!");
```

The output of this code will be similar to the following:

```
2010-05-10 19:09:25 - INFO @ SomeProject.Form1, Text: Form1 - Hello world!
```

Please note that "this.Log..." only works in non static classes. In static classes use "Logger.Log..." instead and provide a sender as parameter if you want to give a context (an example for this is shown further below).

The first line is used to register a logger. All loggers are registered centrally and can be enumerated using Logger.Loggers. A known logger can also be removed again using the Remove method. There are a number of overloads allowing certain kinds of settings to be configured when adding a logger. These will be described further below.

The second line is used to write "Hello World!" to the logging framework. Log messages are usually grouped into classes of severity or other. The following message classes are predefined:

Message Class	Default Severity
STATUS	0
INFO	1
WARN	2
ERROR	3
FATAL	4

The severity is a float value to allow configuring a severity setting in between existing values. But the values can also be overridden and it is furthermore possible to add your own message classes and remove or replace the predefined values as shown below:

```
Logger.RegisterMessageClass<MyMessageClass1>(0.5);
Logger.RegisterMessageClass(typeof(MyMessageClass2), 5);
Logger.MessageClassSeverity[typeof(MyMessageClass3)] = 3.5f;
Logger.MessageClassSeverity.Remove(typeof(INFO));
Logger.MessageClassSeverity[typeof(WARN)] = 100;
Logger.RegisterMessageClass<FATAL>(10);
```

Removing a message class or not providing a severity value doesn't prevent it's usage though. For unknown message classes the framework will assume zero as severity.

It is also possible to prevent the framework from using the defined severity values by setting Logger.UseMessageClassAsSeverity to false. In this case you must explicitely provide a severity value when logging or the severity will be set to zero for the given message:

```
this.Log<ERROR>("Hello world!", Severity => 20f);
```

The expression in the second parameter will be automatically parsed by the framework. Note though that predefined parameters like "Severity" are case sensitive and values do not automatically cast, therefore 20f will work but 20 (int) or 20.1 (double) may not be correctly recognized. Finally, if you provide a Severity value in spite of a predefined value being available, the predefined value will be overwritten.

The logging framework will store the sender as object and the predefined loggers will try to use the sender's ToString() method to retrieve the sender name. However if a sender of type string is provided manually as shown below, the value of sender will be used instead:

```
Logger.Log(MyStaticClass.Instance, "Hello World!");
Logger.Log("MyNamespace.MyStaticClass", "Hello World!");
Logger.Log("Hello World!", Sender => "MyNamespace.MyStaticClass");
this.Log("Hello World!", Sender => "MyNamespace.MyStaticClass");
```

The code in the last line will result in the logger using "MyNamespace.MyStaticClass" as sender no matter what "this" is. As you can see in the examples above it is possible to use the Log function with or without a generic parameter. If no generic parameter is used, the framework will use Logger.DefaultMessageClass (default is INFO) or Logger.DefaultExceptionClass (default is ERROR) if an exception is provided as shown below:

```
this.Log("Hello Exception!", someException);
```

The above line will use "ERROR", though both settings can be changed of course. If – as shown above – an exception and a message are provided, the framework will append the exception message to the given message if Logger.ConcatExceptionMessage is set to true (default is true). The two strings will be separated using Logger.ConcatExceptionMessageSeparator (default is space).

The Logging Process

Normally logging is done in a seperate thread. That means that if you log messages your code will continue to be executed after the message has been passed on to the framework. This may result in messages being lost if your application crashes. However, you can call Logger.Dispatch() at any time to force the framework to process all currently queued messages. Furthermore you can set Logger.AutoDispatch to true (default is false) to force the framework to always process the messages before handing back execution control to your own code. This can result in your application being slowed down but may sometimes still be desirable, especially if the logging is used to provide feedback to the user (for example in splash screens).

If you call Logger.Shutdown() or Logger.Shutdown(false) this will also result in the remaining messages being processed. However you only need to call shutdown if one of your loggers requires this (the predefined loggers do not require a shutdown call). Logger.Shutdown(true) will force the

framework to shutdown the dispatcher thread immediately. This may result in messages being lost.

Adding and Configuring Loggers

When adding a logger to the framework there are a number of ways these can be configured. First of all, using one of the following methods a number of message classes can be provided. The logger will only be notified of these classes:

```
Logger.Add<STATUS>(SomeStatusLogger);
Logger.Add<ERROR, FATAL>(LoggerForBadThings);
Logger.Add(LoggerForHarmlessThings, INFO, STATUS, WARN);
```

Alternatively a minimum severity can be provided:

```
Logger.Add<STATUS>(SomeStatusLogger, 2f);
```

The default minimum severity (used by DefaultLoggerSettings) is one. It is also possible to provide a delegate which will filter messages in it's own way:

```
Logger.Add(SomeBiasedLogger, data => ((string)data["Message"]).Contains("something"));
```

Finally you can provide an arbitraty ILoggerSettings instance. If you do not want to implement your own logger settings class you can use DefaultLoggerSettings which allows roughly the same settings as the above methods. Otherwise you just have to Implement ILoggerSettings which only requires one method.

To create your own loggers, use the ILogger interface.