**Handlers and Formatters**

A Handler typically uses a Formatter to format the message before logging it. You can either create your ownFormatter or use one of the two built-in Formatter's. To learn more about Formatter's see the text on[**Formatters**](http://tutorials.jenkov.com/java-logging/formatters.html).

Here is how you get and set a formatter on a Handler:

ConsoleHandler handler = new ConsoleHandler();

handler.setFormatter(new SimpleFormatter());

Formatter formatter = handler.getFormatter();

For some built-in Handler's the formatter can also be set via the configuration file. See the text on **[configuration](http://tutorials.jenkov.com/java-logging/configuration.html)**for more details.

**Built-in Handlers**

You can create your own Handler if you want, but Java comes with 4 built-in Handler's already:

1. ConsoleHandler
2. FileHandler
3. StreamHandler
4. SocketHandler
5. MemoryHandler

You will most often use the FileHandler, but each of these built-in Handler's are explained briefly in the following sections.

**ConsoleHandler**

The ConsoleHandler logs all messages to System.err. By default the ConsoleHandler uses aSimpleFormatter to format the messages before writing them to System.err. Here is how you create aConsoleHandler:

ConsoleHandler handler = new ConsoleHandler();

**FileHandler**

The FileHandler writes all messages to file. This can either be a single file, or a set of rotated files. If rotated files are used, each file is filled to a certain size limit, after which a new file is created. Each file name is composed of a base name and a sequence number. For instance mylog.0.txt, mylog.1.txt etc.

By default the FileHandler uses the XMLFormatter to format all messages before writing them to a file.

Here are the various constructors you can use to create a FileHandler:

FileHandler handler = new FileHandler();

FileHandler handler = new FileHandler(String pattern);

FileHandler handler = new FileHandler(String pattern, boolean append);

FileHandler handler = new FileHandler(String pattern, int limit, int count);

FileHandler handler = new FileHandler(String pattern, int limit, int count,

boolean append);

The first constructor creates a default FileHandler. This FileHandler is fully configured via the configuration file.

The second constructor creates a FileHandler with a predefined pattern for generating file names for the log files.

The third constructor creates a FileHandler with a file name pattern, and a boolean telling whether theFileHandler should append to any existing files or not. There is no file size limit, and file count is set to 1.

The fourth constructor creates a FileHandler with a file name pattern, a file size limit, and a file count. When the log files reach the given file size limit a new file is created, until the maximum of the file count is reached. Then theFileHandler starts over with the first file again, deleting it and logging to it from scratch.

The fifth constructor creates a FileHandler with a file name pattern, a file size limit, a file count, and a boolean telling whether the FileHandler should append to any existing files or not.

Here are a few examples:

FileHandler handler = new FileHandler("myapp-log.%u.%g.txt");

FileHandler handler = new FileHandler("myapp-log.%u.%g.txt",

true);

FileHandler handler = new FileHandler("myapp-log.%u.%g.txt",

1024 \* 1024, 10);

FileHandler handler = new FileHandler("myapp-log.%u.%g.txt",

1024 \* 1024, 10, true);

**File Name Pattern**

The file name pattern is a string containing a file name plus one or more special codes, telling the FileHandlerhow to generate the file names. The special codes you can use are:

|  |  |
| --- | --- |
| Code | Meaning |
| / | The file name separator of the system. Typically either \ or / . |
| %t | The temp directory of the system. |
| %h | The user home directory of the system. |
| %g | The generation number that distinguishes the rotated log files from each other. |
| %u | A unique number to avoid naming conflicts. |
| %% | A single percent sign, in case you want to use that in your file name. |

If no %g code has been specified and the file count of the FileHandler is greater than 1, then the generation number (file sequence number) will be appended to the end of the file name, after a dot (.) .

The %u code is usually set to 0. If a file already exists with that name which is in use by another process, the %u code is set to 1, 2 etc. until an unused base file name is found. If no %u code is used in the file name pattern, and a conflict over the file name is found with another process, the unique number is appended to the end of the file name, after any automatically added generation number. Note: The use of %u to generate unique numbers for base file names is only guaranteed to work on a local file system.

Here are a few examples:

|  |  |
| --- | --- |
| File Name Pattern | Meaning |
| logfile.txt | The file is called logfile, and is located in the current directory of the application. |
| logfile%g.txt | The file is called logfile, and is located in the current directory of the application. A generation number is inserted after the text "logfile" in the file name. For instance,logfile0.txt, logfile1.txt etc. |
| logfile%u.%g.txt | The file is called logfile, and is located in the current directory of the application. A unique number and a generation number is inserted after the text "logfile" in the file name. For instance, logfile0.0.txt, logfile0.1.txt etc. |

**StreamHandler**

The StreamHandler writes the log messages to an OutputStream. Here is an example of how you create aStreamHandler:

StreamHandler handler = new StreamHandler();

StreamHandler handler = new StreamHandler(outputStream, formatter);

The first constructor creates an empty StreamHandler with no OutputStream. You must set one using the method setOutputStream() before using the handler.

The second constructor creates a StreamHandler with an OutputStream and a Formatter TheoutputStream parameter should be some OutputStream you either create or obtain from some other object. For instance, it could be a FileOutputStream.

The Formatter should be whatever formatter (built-in or custom implementation) you want to use. TheStreamHandler uses a SimpleFormatter by default (if you use the zero-arg constructor).

**SocketHandler**

A SocketHandler writes the log messages to some network address via a socket. The log messages are sent across the network raw (as text). They are not wrapped in an HTTP request or anything like that.

Here is how you create a SocketHandler:

SocketHandler socketHandler = new SocketHandler(host, port);

The host parameter should be a string pointing to a domain name (host). The port parameter should be the TCP port to connect to.

Here is an example that uses concrete values in the constructor:

SocketHandler socketHandler = new SocketHandler("jenkov.com", 80);

**MemoryHandler**

A MemoryHandler is a handler that keeps the LogRecords internally in a buffer. When the internal buffer is full, new LogRecords start overwriting the oldest ones in the buffer.

When a certain trigger event occurs, the LogRecord's in the internal buffer are flushed to a target Handler which will write the LogRecords to an external system. For instance, when a LogRecord of some minimum log level is logged, the whole buffer of LogRecord's could get pushed.

You can also call the push() method to force the LogRecord's in the buffer to be flushed to the target Handler.

Here are two examples creating a MemoryHandler:

MemoryHandler handler = MemoryHandler();

MemoryHandler handler = MemoryHandler(

targetHandler, bufferSize, pushLevel);

The first constructor creates a MemoryHandler that uses the configuration obtained from the configuration file.

The second constructor creates MemoryHandler with a target Handler, a specific internal buffer size, and a minimum push level that gets LogRecord's pushed to the target Handler.

Here is an example using concrete values for the buffer size and push level parameters in second constructor:

MemoryHandler handler = MemoryHandler(

targetHandler, 10, Level.WARNING);