Pattern Conversion Characters

Let's see the following table describes the characters used in the conversion pattern and all other characters that we can use in our custom pattern:

|  |  |
| --- | --- |
| **Conversion Character** | **Meaning** |
| c | It is used to output the category of the logging event. For Example: for the category name x.y.z the pattern %c{2} will output y.z. |
| C | It is used to output the fully qualified class name of the caller issuing the logging request. For example, for the class name "org.apache.abc.MyClass", the pattern %C{1} will output "MyClass". |
| d | It is used to output the date of the logging event. For example, %d{HH:mm:ss,SSS} or %d{dd MMM yyyy HH:mm:ss,SSS}. |
| F | It is used to output the file name where the logging request was issued. |
| l | It is used to output location information of the caller which generated the logging event. |
| L | It is used to output the line number from where the logging request was issued. |
| m | It is used to output the application supplied message associated with the logging event. |
| M | It is used to output the method name where the logging request was issued. |
| n | It is used to give the output of platform-dependent line separator character or characters. |
| p | Outputs the priority of the logging event. |
| r | It is used to output the number of milliseconds elapsed from the construction of the layout until the creation of the logging event. |
| t | It is used to output the name of the thread that generated the logging event. |
| x | It is used to output the NDC (nested diagnostic context) associated with the thread that generated the logging event. |
| X | The X conversion character is followed by the key for the MDC (Mapped Diagnostic Context). For example, X{clientIP} prints the information stored in the MDC against the key clientIP. |
| % | The literal percent sign. %% will print a % sign. |

**Log4j has the following levels of logging:**

|  |  |
| --- | --- |
| **Log Level** | **Description** |
| ALL | This level turns on all levels of logging. It includes the custom logging levels that you have defined. Once this one is configured and the levels are not considered at all, then all the appenders will start pouring the log events in log files. |
| DEBUG | Debug is used a lot for debugging the application at development time. Every log message will appear to log files once this level is set. It basically belongs to developers. |
| INFO | The INFO logging level is used to record messages about routine application operation. In real-time, system administrators watch the info logs to ensure what's happening on the system right now, and if there is any problem in normal flow. |
| WARN | WARN log level is used to indicate that you might have a problem and that you've detected an unusual situation. Maybe you were demanding to invoke a service, and it failed a couple of times before connecting on an automatic retry. It is unexpected and unusual, but no real harm was done, and it's not known whether the issue will persist or recur. Someone should investigate warnings. |
| ERROR | The ERROR log level is used to denote a serious problem that you must have to investigate immediately. Not as serious as FATAL, but still a problem. It simply means that your application has met really undesired state. For example, unexpected formatted input, database unavailability. |
| FATAL | The FATAL log level, like ERROR, designates a problem. But unlike ERROR, it designates a very serious error event. You will not consider their presence very much on a normal day, but once they appear, it signals very bad news, even the application of death. |
| OFF | This is the highest possible rank and is intended to turn off logging. |
| TRACE | This has been recently introduced in version 1.2 and includes more information to debug level logs. |

Let's consider that as the following rank order for the levels:

1. ALL **<** **TRACE** **<** **DEBUG** **<** **INFO** **<** **WARN** **<** **ERROR** **<** **FATAL** **<** **OFF**

