Tutorial – Logging & Tracing Mechanism in SAP

Version 1.2

Infrastructure Management Team

Table of Contents

Table of Figures

1. About this tutorial

This tutorial covers the new logging and tracing framework (referred as ‘logging API’ or ‘SAP Logging API’ hereafter), provided for the Java development projects at SAP. The tutorial focuses on *how to use* the logging API. The technical implementation details are not covered. Step-by-step instructions and examples are provided. The goal is to encourage and enable the readers to readily start using the logging API after finishing this tutorial. Special features made available in this tool are also introduced in details.

It is recommended to go over the basic concept of the logging framework, as described in **Chapter 2 and 3**. But for those developers who want to skip the basics and look at some typical logging operations right away, you may do so by focusing on **Chapter 4.2** and **4.6**. Other useful basic and advanced properties are covered in **Chapter 4.4 and 4.5** respectively. A few administrative topics and the outlook on future development are talked about in **Chapter 5 and 6**.

1.1 Target Audience

Primarily all SAP Java developers who are required to provide logs in a productive environment for the customers, or who are simply interested in having traces to assist their own debugging tasks are the audience.

1.2 Things you should know in advance

Prerequisites? In short: Nothing.
Certainly, we expect the developers, who are responsible to instrument the logging message in their code, have a common sense about where and what to log intelligently. Other than this, enabling traces or logs is fairly straightforward.
Only when you attempt to use more advanced features, for example, language-neutral message, you will need to understand the concept of `ResourceBundle` in Java. (Please refer to Section 4.4.7 for more details about advanced features)

2. Introduction

2.1 What is Logging and Tracing?

During the development phase, it is a common practice of developers to insert logging statement within the code to produce informative logs at the runtime, whether for troubleshooting or for analysis purposes.

Common problem
Without a standard logging framework, developers are very likely to use `System.out.println` in a sporadic manner or call `printStackTrace()` in case of an exception. Before executing the code in a productive system, the cleanup of these embedded lines can be very painstaking. Apparently, this is not very flexible in terms of controlling the amount of log output, the destination, and the format of the messages, etc.

Two main types
In our logging API, we would even like to further break down the messages into:

- Logging: classical log messages, for distinguished problematic areas
- Tracing: classical trace messages, for coding problems

2.1.1 Purpose

There are various infrastructure and application development groups in SAP coding in Java, and either they have their own (quick & dirty) logging mechanism, or they do not do this at all. However, the availability and readability of traces or event logs are very important for both the developers and the potential users, like testing group, support group and operators. Therefore, a common logging framework must be made available to satisfy the requirements of these groups, especially in a large and distributed environment. A standardized infrastructure will be essential and beneficial for both the developers and the consumers.

2.1.2 Advantage

In designing this framework, we have kept the following in mind:

- Easy to use API
  Enabling logs is not a popular task for developer. So, the API/method calls to do so have to be simple and intuitive.
- Performance
  The switching on of the logging mechanism should not degrade the performance of the application as if running with logging,
- Easy to maintain
  The log insertion done by developer is totally decoupled from executing the code. Switching on or controlling the amount of log output is configurable at runtime without modifying the source code.

2.2 Current Package

- *com.sap.tc.logging* – referred as 'SAP Logging API', with all functionality for both tracing and events logging

2.2.1 Overall Logic

General procedures:

1. Identify the source area you would like to produce trace/log output
2. Assign severity level to the source
3. Specify output destination
4. Insert messages with corresponding severity level

Run the program.

When is the message produced?

Only when the severity of the message is equal or higher than the source, the message will be produced and sent to the destination.

Main tasks

From a developer's point of view, normally, you will simply focus on step 1 and 4, which are, logging meaningful messages at good points of the execution flow. The other 2 steps are more or less determined and configured by the operators/end-users eventually. This shows the idea of decoupling the task of enabling the logs by the developers and the program execution by the users.

Sample idea and output

For example (corresponds to the steps mentioned above):

1. Source area: a class '`fooClass`' under an arbitrary package '`com.saps.fooPackage`'
2. Severity of the source: *Debug*
   - Allow messages with severity *Debug* level or above to be printed
3. Destination: a file
4. Insert an informational text at the very first line of a method, e.g. `fooMethod()`, indicating the entry point to this method, with severity *Path* (higher than *Debug*)

The message text will be written to the file with a standard format as shown below:

```
May 3, 2001 6:54:18 PM   com.sap.fooPackage.FooClass.fooMethod [main]  Path: Entering method
```

This corresponds to:

Date and timestamp   Full path method name [thread name] Severity: Message

2.2.2 Conceptual view and terminology

Figure2-1 shows a simplified version of the conceptual view that shows the key entities defined in the SAP Logging API. Brief introduction of the terms is given in Table2-1.

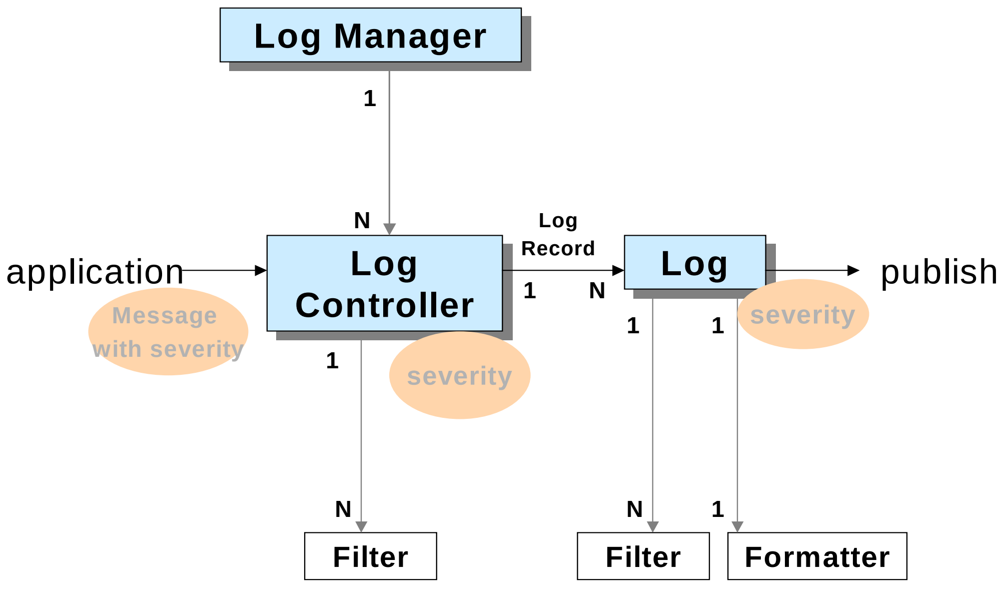


Figure 2-1 Conceptual View of SAP Logging tool

Term	*Description*
Logging Manager	A unique single manager that manages mainly the log controllers
Log Controller	Entity that represents the source area
Log Record	Structure that holds a message and its relevant data
Log	Represents the destination where the message should be output to
Formatter	Determines the format of the final message text
Filter	Optional means to further screen out messages

Table 2-1 Terminology of key components

3. SAP Logging API Basics

3.1 Background

The concept of the SAP logging tool is very similar to the other available tools in the market. It offers better support for a common practice in logging:

- Generates classical 'trace messages' and 'log messages'
- There exists APIs that are handy to bridge these two types of messages together for advanced analysis.
- Other advanced features will be covered in the later sections.

Are you outputting messages for logging or for tracing?
If your goal is to pinpoint distinguished problematic logistic areas, then you should focus on 'log messages', while if you are more interested in analyzing coding problems, then 'trace messages' is the one to be used.
Yes, this will be the major task as a developer to enable logging: classify the source area that needs to be logged/traced, and then insert the output messages in the code. Simple enough.

What are the basic classes to start with?
From the conceptual view diagram Figure 2-1, you should know by now the Java class that represents the source area: *Log Controller*. In general, you deal with its two subclasses directly:

- *Category*: generate 'log messages'
- *Location*: generate 'trace messages'

Examples for a *Category*:
"/System/Database", "System/Security", …
Examples for a *Location:*
"com.sap.tc", "com.sap.tc.logging", …
(Please refer to **Appendix B** for a complete class hierarchy diagram defined for this logging API).

Let's start getting familiar with the tool by focusing on one area. Since both API for logging and tracing are quite similar, it will be very straightforward to use one or the other when you have learnt one of them. Most examples will be focused on the 'tracing' API.

3.2 Common logging methods

With the *Location* or *Category* defined, you are ready to insert output methods in your code to fire messages whenever necessary. Each of the call is already assigned a severity level and this provides a clean and easy API for users to use readily.

There are a number of methods available to write messages with different severity. They loosely fall into two groups, where the first group has intuitive name with severity level indicated. These methods produce messages with the respective severity incorporated in the method name. According to the severity scale shown in **Appendix A**, you can find, for example with *Location* class:

Common output API with severity indicated
fatalT(string the_message)

errorT(string the_message)
warningT(string the_message)
infoT(string the_message)
pathT(string the_message)
debugT(string the_message)

Table 3-1 Common Output API – group 1

The second group also has intuitive names, but without explicit severity level shown in their names. Table 3-2 below shows the ones that are commonly used:

Common output API (mostly for program flow)	***Description***
entering()	Denote method entry with level *Severity.Path*. Always used together with *exiting()*.
exiting()	Denote method exit with level *Severity.Path.* Always used together with *entering()*.
throwing(Throwable the_exception)	Output the exception content with level *Severity.Warning*
assertion(Boolean the_assertion, String the_message):	A trace method to verify your assertion and will throw a *Severity.Error* message when the condition is false.

Table 3-2 Common Output API – group 2

Refer to the Javadocs, it may first appear that there are overwhelmingly numerous methods, but they are pretty much overloaded methods with different arguments to enhance flexibility. It is better to provide more options for different requirements, rather than inadequate APIs. (More details shown in Section 4.2.4).

The APIs shown above are in their simplest form for clarity. But even with these methods, you are already in good shape to start playing with logging/tracing.

3.3 Simple Example Flow

1. Identify the source area and get a handle to a source object: *Location*
   - Name the 'id' of the source *location* object as the complete package path of the class, e.g. `'com.sap.fooPackage.FooClass'`
2. (Assign severity level to the *location* object)
3. (Specify output destination for the *location* object)
4. Insert messages at points where you want to trace the program flow with desired severity level
   - The common places are: entering/exiting a method, upon throwing an exception, after performing certain critical tasks, etc…
   - Decide the severity to be assigned to these trace messages.

3.3.1 Tracing

```
package com.sap.fooPackage;

public class FooClass{

   public void method1(){
```

A typical program without tracing inserted.

```
            ………
        }

        public void method2(){
           try{
                 …….
            }
                     catch (IOException e){
                 ……..
                 }
        }

        public static void main(String[] args){
           FooClass test1 = new FooClass();
           test1.method1();
           test1.method2();
         }

    }
```

Now, let's insert traces into the program (original code is grayed out for your convenience). The API to create a source trace object is simple, as well as the output methods used for writing traces.

Sample program showing the instrumentation of traces

```
    package com.sap.fooPackage;

    // (0)
    import com.sap.tc.logging.*;

    public class FooClass{

    // (1) The current java class is defined as a location source
      private static Location myLoc =
      Location.getLocation("com.sap.fooPackage.FooClass");

    //  (2) Assign severity, otherwise, nothing will be printed out by default
    //      Assign destination, no destination by default
       static{
         myLoc.setEffectiveSeverity(Severity.PATH);
         myLoc.addLog(new ConsoleLog());
       }

       public void method1(){
          // (3) Enable the writing of trace messages
         myLoc.entering("method1");
             ………
         myLoc.warningT("method1", "Sample warning message");
             ………
         myLoc.exiting();
       }

       public void method2(){
         myLoc.entering("method2");
          try{
                 …….
          }
          catch (IOException e){
            myLoc.throwing("method2", e);      //e.g. file not found exception
                 ……..
          }
         myLoc.exiting();
        }

       public static void main(String[] args){
         FooClass test1 = new FooClass();
         test1.method1();
         test1.method2();
       }

    }
```

3.3.2 Explanation

There are mainly a few additional lines you need to activate the tracing feature. Refer the numbering below to that in the sample source code.

- (**0**): Import the SAP logging package
- (**1**): Get access to a *location* object for the current class. The API provided should be intuitive enough: a class method provided by *Location* class. Users do not have to work on constructor, just call `Location.getLocation(<name of location object>)`, and receive a handle to the unique *location*
- (**2**): Assign severity level and destination for the *location* object
  This will be the level to be compared against the severity of the trace messages. Messages with severity lower than this of the *location* object will not be printed. Refer to **Appendix A** to get familiar with the severity level definition.
  By default, *Severity.NONE* will be assigned (that is, there will be no output at all). So, for this sample coding, an explicit level *Severity.PATH* is assigned to screen out all the debugging messages, which have a lower severity.
  Also by default, no output destination is assigned (again, no output at all). In the example above, we direct the output to the console. Refer to **Appendix B** with class diagram that shows that we mainly have two types of *logs*: *FileLog*, *ConsoleLog*.
  (In Section 4.6, you will learn that it is not necessary to hardcode it here in the source, and you can configure this setting externally without messing with the code.)
- (**3**): Insert trace messages in your program
  The example shows a few APIs that write out trace messages`: entering, warningT, throwing, exiting`. Which messages will eventually be printed? We know that all these messages will successfully pass the level *Severity.PATH* (as specified in this example in step (2)).

3.3.3 Output

In the example above, messages will be directed to the console with the (default) format readable by operators (using *TraceFormatter*).

The content of the output file for the example above looks like this:

```
May 3, 2001 6:54:18 PM com.sap.fooPackage.FooClass.method1 [main]
                 Path: Entering method
May 3, 2001 6:54:18 PM com.sap.fooPackage.FooClass.method1 [main]
                 Warning: Sample warning message
May 3, 2001 6:54:18 PM com.sap.fooPackage.FooClass.method1 [main]
                 Path: Exiting method
May 3, 2001 6:54:18 PM com.sap.fooPackage.FooClass.method2 [main]
```

```
                                     Path: Entering method
           May 3, 2001 6:54:18 PM com.sap.fooPackage.FooClass.method2 [main]
                                     Warning: Throwing java.io.FileNotFoundException:
                                     C:\Not_Exist\zzzzz.log (The system cannot find the path
                                     specified)
           May 3, 2001 6:54:18 PM com.sap.fooPackage.FooClass.method2 [main]
                                     Path: Exiting method
```

3.4 Coding Recommendations

The logic so far is straightforward and the learning curve is not steep, just a few additional lines, and you can enjoy the decent result in a log file.

Before we wrap up this section, there are certain recommendations for you to improve the tracing functions.

- **Static reference**
  As you may have noticed from the sample coding, the variable that holds the *location* object is declared with a *static* modifier to improve efficient access. Basically, a handle to a *location* object can always be obtained by calling `Location.getLocation(<id>)`. However, making such a call every single time may degrade the performance.

- **Hierarchical naming convention**
  It is a good idea to create a static variable for the trace object and it is a useful and easy approach to name the <*name of the location object*> as the complete package path of the class, for both clarity and the inheritance features made available in the logging API.
  When naming a *location* object for your Java class, it is a straightforward approach to use a hierarchical naming convention, e.g. the fully qualified name. This not only avoids ambiguity on classes having the same name, but also takes the full advantage of the inheritance feature (e.g. on severity level and output destination) implemented by the logging framework.
  The class *`com.sap.foo`* is parent to classes *`com.sap.foo.classA`* and *`com.sap.foo.classB`*, and the latter two classes are siblings to each other.
  In this case, severity level and logs assigned to parent *`com.sap.foo`* will become effective for its children *`com.sap.foo.classA`* and *`com.sap.foo.classB`*. Siblings will not affect each other.

  Some developers may argue about the potential typing error of a lengthy correct class name that will affect the inheritance features. That is true. There exists another API for `Location.getLocation(<java class>)` which accepts a class other than a string. Please refer to the JavaDoc for an API that fits your requirements.

4. SAP Logging API (Detailed explanation)

4.1 Default Setting

In the previous section, you have already encountered a few times that a default value has already been set and you simply have to accept the default behavior mentioned in the example. These settings can be easily manipulated and changed by you through the API or an external configuration file (details in Section 4.6).

Advantages

Default values are meant to provide a reasonable logging behavior with minimal configuration required to be done by the users. For example, once a user has specified to output the message to a console, it is assumed to be read by operators, and thus, a *TraceFormatter* (formatting message that is readily understood by human being) is selected with the predefined message pattern that can clearly displays the message and its relevant info:

```
May 3, 2001 6:54:18 PM   com.sap.fooPackage.FooClass.fooMethod [main]
                         Path: Entering method
```

References

There will be more predefined values appearing in the following sections. To assist you in moving on smoothly with the tutorial, as well as avoiding ambiguity for your future coding, refer to **Appendix C** where we are keeping track of the various default settings and behavior.

4.2 Detailed Example

To recap again, there are mainly 4 steps in enabling logging for your application.

- Identify source code area, e.g. a class 'Node', and represent it with a *Location* object
  `Location loc = Location.getLocation("<package>.Node");`
- Assign severity level to the source
  `loc.setSeverity(Severity.WARNING);` //default: *Severity.NONE*
- Specify output destination
  `loc.addLog(new ConsoleLog());` //with default formatter: *TraceFormatter*
- Instrument trace message with specified severity
  ```
  loc.entering(methodname);
  loc.debugT(methodname, message);
  loc.fatalT(methodname, message);
  ```

It is a legitimate deduction that more than 90% of your development time will be spent on step 1 and 4. Eventually, step 2 and 3 can be mainly configured externally so that users can control the output without the need to modify source code and recompile again.

Shown below is a very basic sample code, showing step 1 & step 4:

```
    package com.sap.fooPackage;

    import com.sap.tc.logging.*;

     public class Node {
       private static final Location loc =
             Location.getLocation("com.sap.fooPackage.Node");

       public void announce(Object o) {
          String method = "announce(java.lang.Object)";
```

```
          loc.entering(method);
          try{
            // do something...
            loc.debugT(method, "Connecting to …");
          }
          catch (Exception e) {
            loc.fatalT(method,
                       "Error processing object {0}",
                       new Object[] {o});
          }
          loc.exiting();
        }
      }
```

Step 2 & 3 are not shown at this point, but assuming the severity level assigned to this is set to be *Severity.ALL* (accept all severity levels and output everything) and output has been piped to a *ConsoleLog* (terminal).
The output will look like this, formatted with *TraceFormatter* (default formatter for *ConsoleLog*).

```
May 3, 2001 6:54:18 PM com.sap.fooPackage.Node.announce [main]  Path: Entering method
May 3, 2001 6:54:18 PM com.sap.fooPackage.Node.announce [main]  Debug: Connecting to ….
May 3, 2001 6:54:18 PM com.sap.fooPackage.Node.announce [main]  Path: Exiting method
```

The following 4 sections will further explain each step in details. Bear in mind that in this section 4.2, we try to focus only on the most basic scenario and its API and avoid jumping into the advanced logic. This should be good enough to get you started. But you will find frequent references to later sections that cover the logic or advanced features in details.

4.2.1 Identify the output source: *Location* or *Category*

Both *Location* and *Category* are subclasses of *LogController*.

- *Location* is the source area that generates trace messages. Typically, it corresponds to the source code structure, and can be attached to the level of component, package, class or method.
- *Category* is the source area that generates log messages, corresponding to a distinguished problem area, such as networking, database, …

Naming of the source
Although the naming of a *Location* and *Category* is quite flexible, the normal rule is: a valid hierarchical naming convention.

E.g. *Location*: com.sap.fooPackage.Class1
com.sap.fooPackage.Class2 (make sure there is no typo)
Category: /System/Networking
/System/Database (always starts with '/')

A common naming practice for *Location* is, name the *location* exactly with the full path of Java package name. Certainly you can randomly name a *location* with 'testLocation1_class1' for the class `'com.sap.fooPackage.Class1'`, but the output will not be very meaningful.

Various access API
A static method is provided for each class for easy access to a *location* or *category*:

```
Location.getLocation(<name of the Location>);
Category.getCategory(<name of the Category>);
```

Or instead of passing the name manually, for *Location,* you may as well pass the class instance (`java.lang.Object`) or the class itself (`java.lang.Class`). Refer to the Javadocs for details. In this case, the *location* object is by default referring to the class level, while using the string argument `(java.lang.String)`, you have the flexibility in the definition, e.g. include also the method name to explicitly control logging over methods individually.

Now that you have got a handle to the source, you are ready to configure and ask this source to generate messages. It is recommended that initially, you should assign make the handle to be **static** to improve the efficiency:

```
  static final Location loc = Location.getLocation(this.getClass())
```

4.2.2 Assign severity to source

Again, refer to **Appendix A** for various severity level definitions. Simply make use of the constants provided directly: *Severity.DEBUG, Severity.PATH, Severity.INFO, Severity.WARNING, Severity.ERROR, Severity.FATAL, Severity.ALL* and *Severity.NONE.*

Normally, you assign severity to the source by:

```
        loc.setEffectiveSeverity(Severity.INFO);
```

Then, any messages with severity lower than *INFO* will be discarded, others will be directed to the destination.

Introduce concept of hierarchical severity

Hierarchical severity will be mentioned in Section 4.4.2, but one quick note is, because of the hierarchical naming feature of *location* or *category*, you can save some effort by assigning severity to the parent, and its children will automatically inherit the assigned severity as well.

```
  static final Location loc = Location.getLocation(com.sap.fooPackage);
  loc.setEffectiveSeverity(Severity.INFO);
```

If you have a *location* '`com.sap.fooPackage.Class1`', it will have inherited the severity *INFO* already.

Default with strictest severity

By default, source object (assume the ascendants has not been assigned severity yet) has SEVERITY.NONE. Therefore, developer can freely enable the output methods in the source code, but the actual logging is not activated until it is explicitly 'switched on' when things are ready.

4.2.3 Specify output destination

We need output destination to print out the messages. It is collectively called *Log* in our tool. Intuitively, you assign a *Log* to a *Location* or *Category*. Otherwise, without a *log*, even though you would have set the severity and inserted the output methods correctly, nothing will be printed. You can assign a *log* to a source by using:

```
        loc.addLog(<log1>);
```

Types of *log*

Currently, there are 2 types of logs: *ConsoleLog, FileLog.*
ConsoleLog is for printing to terminal, while *FileLog* is for writing messages to files.

Multiple destinations

There are times you may want to assign multiple *logs* to a single source. Message generated from a source will be sent to both of them simultaneously. For example, output messages to both the console and a file.
The simplest ways to do so is as below*:*

```
loc.addLog(new ConsoleLog());
loc.addLog(new FileLog("C:\\temp\\testOutput.log");   // filepath has to be valid
```

Creating a *log*
As you can see from the Javadoc API, the constructors are overloaded where you can specify certain options for the logs, while we only show the most basic one here (using minimal parameters for simplicity). Therefore, default settings are used and worth noted:

- *ConsoleLog* is using a *TraceFormatter* by default.
- *FileLog* is using *ListFormatter* and the output file will not be a rolling file type, but one single output file, increasing in size when more and more messages appended to it. More configurations can be specified for a *FileLog* and the details are described in 4.4.5.

For example, if you want to switch to a *XMLFormatter* of a *Filelog*, you may call:

```
loc.addLog(new FileLog("C:\temp\testOutput.log", new XMLFormatter());
```

or with an existing log:

```
<filelog>.setFormatter(new XMLFormatter());
```

Attaching a *log*
Refer to the Javadoc, there are two other APIs in assigning *Logs* to a *Location* or *Category*, '`addPrivateLog`', '`addLocalLog`', because there are actually three forms of log assignments. They behave differently in terms of inheritance by children of a *Location* or *Category* source. In short, these three are **mutually exclusive**, and the example '`addLog`' shown here is straightforward for beginners. It supports forced inheritance to children and message output unconditionally. More details can be found in sections 4.4.3 and 4.5.1.

4.2.4 Enable output messages

Now that we have the source defined, severity assigned and destination specified properly, you are ready to insert the output statements in the appropriate places in the code.

Abundant output method APIs
Do not get overwhelmed by the seemingly endless list of output methods. They are not randomly overloaded, but are expanded from the few fundamental ones as described in Section 3.2. You will soon realize the pattern in overloading the signatures is fairly systematic. They are simply heavily overloaded to provide flexibility for the users.

Plus, the set of output methods are very similar between *Location* and *Category*.

At this stage, we will not introduce all the overloaded methods, but focus on a few basic and practical one. (The rest involves better understanding of the logging features, and these are explained in Section 4.5.2).

Let's divide the output methods into 3 main groups for the ease of explanation:

- Typical message output with severity
- Denoting the flow of program
- Master gate

4.2.4.1 Message output with severity

- *Location*
  - `fatalT, errorT, warningT, infoT, pathT, debugT`
- *Category*
  - `fatalT, errorT, warningT, infoT`

The names are self-explanatory about the severity level of the messages generated by these methods. The overloaded pattern is the same for each severity output (masked with ‘xxxx’ here):

Location	***Category***
xxxxT(String message)	xxxxT(Location loc, String message)
xxxxT(String subloc, String message)	xxxxT(Location loc, String subloc, String message)
xxxxT(String message, Object[] args)	xxxxT(Location loc, String message, Object[] args)
xxxxT(String subloc, String message, Object[] args)	xxxxT(Location loc, String subloc, String message, Object[] args)

There exists a pattern in method overloading: evolves around the core argument: *message*. The addition of *subloc*, *args* offers the flexibility for developers to log messages in the level of details that they need. Understanding these arguments can help you select the heavily overloaded methods easier.

loc:
It is obvious that the only difference in the API between *Location* and *Category* is an additional *loc* argument in *Category* output methods. It is a typical request that log messages are always written with respect to a source code area. This proves to be very helpful for logging analysis. By specifying the *loc* argument, you indicate that the message should be written as a trace message associated with the *loc* object. With a little configuration done for *loc*, logging can be just done once but will be piped for both message types (logs & traces) simultaneously. This is explained in details in Section 4.5.2. Vice versa, this works for *Location* as well, and API is available to specify the *category* argument. But since this is optional for *Location*, so we won’t list them here. (Again, refer to Section 4.5.2).

***subloc*:**
Treat the *subloc* argument as the method name of the source class that the message is generated from. This is optional, but with this argument included in the trace/log, the picture when doing analysis will be much clearer, especially you can specify different arguments for overloaded methods.

message:
The actual message to be printed is put in argument *message*. Make up something simple that is meaningfully describing the situation/problem. (Aforementioned, language independency is supported, using ResourceBundle. Refer to 4.4.7 for more examples of output methods).

***args**:*
Array of additional arguments that are informative, e.g. dynamic information to be included in the message. This is achieved by using `java.text.MessageFormat` API to resolve arguments.

To reiterate, refer to the sample code at the beginning of Section 4.2.3, with method *announce(Object o)*. This is a simple example of working on *Location* object.

```
package com.sap.fooPackage;

import com.sap.tc.logging.*;

public class Node {
  private static final Location loc =
          Location.getLocation("com.sap.fooPackage.Node");

  public void announce(Object o) {
    String method = "announce(java.lang.Object)";
    try {
    // do something...eg. connecting to DB, perform certain actions
      loc.debugT(method, "Connecting to ….");
      //minor error in writing something
      loc.warningT(method,
                   "Problems in row {0} to {1}",
                   new Object[] {row1, rowN});
      //finish successfully
      loc.infoT(method, "DB action done successfully");
    }
    catch (Exception e) {
    }
  }  // method announce
}  // class Node
```

Potential output, assuming the simplest case with ConsoleLog and default TraceFormatter:

```
May 3, 2001 6:54:18 PM com.sap.fooPackage.Node.announce [main] Debug: Connecting to ….
May 3, 2001 6:54:18 PM com.sap.fooPackage.Node.announce [main] Warning: Problems in row 15 to
18
May 3, 2001 6:54:18 PM com.sap.fooPackage.Node.announce [main] Info: DB action done successfully
```

Another example of working on *Category* object:

```
package com.sap.fooPackage;

import com.sap.tc.logging.*;

public class Node {
  private static final Location loc =
          Location.getLocation("com.sap.fooPackage.Node");
  private static final Category cat =
          Category.getCategory("/System/Database");

   public void store() {
     try {
      // Write object data to database ...
     }
     catch (FailedRegistrationException e) {
       cat.errorT(loc,
                  "store()",
                  "Error storing node {0} in database.",
                  new Object[] {this});
     }
```

```
        }  // method store
      }  // class Node
```

Note that the output will be identical to the previous example, assuming the default setting is used (with *ConsoleLog* and default *TraceFormatter*). Only when you configure the output display accordingly (e.g. use another formatter, or change the pattern of the TraceFormatter), then you will get different result. See Section 4.4.1 for more information.

4.2.4.2 Program flow

This is **only** available in *Location.* Tracing the flow of program is a common practice: `entering, exiting, throwing, assertion.`

- entering: output a default message ("Entering Method" with *Path* severity) indicating that it is entering a source block. **NOTE: ALWAYS paired up with method '*exiting*'.**

Method	***Description***
entering()	Entering a source block in general
entering(String subloc)	Specify the method name in *subloc*
entering(Object[] args)	A general source block with arguments: "Entering method with <args>"
entering(String subloc, Object[] args)	Same as above but with specific method name

- exiting: output a default message ("Exiting Method" with *Path* severity) indicating that it is leaving a source block. **NOTE: ALWAYS paired up with method '*entering*'.**

Method	***Description***
exiting()	Exiting a source block in general. As long as the methodname (subloc) is specified in '*entering*' , it is not necessary to provide *subloc* as argument here anymore. See the result of the following sample code.
exiting(String subloc) //DEPRECATED	Specify the method name in *subloc*
exiting(Object res)	A general source block with result: "Exiting method with <res>"
exiting(String subloc, Object res)	Same as above but with specific method name

To reiterate, refer to the sample code with method *announce(Object o)*:

```
    public void announce(Object o) {
      String method = "announce(java.lang.Object)";
      loc.entering(method);
      try {
      }
      catch (Exception e) {
      }
      loc.exiting();
    }
```

Always log 'entering' & 'exiting' methods together.

Potential output, assuming the simplest case with ConsoleLog and default TraceFormatter:

```
May 3, 2001 6:54:18 PM com.sap.fooPackage.Node.announce [main] Path: Entering method
May 3, 2001 6:54:18 PM com.sap.fooPackage.Node.announce [main] Path: Exiting method
```

- throwing: warning message (“Throwing …”) indicating that the source block is about to throw an exception

Method	***Description***
throwing(Throwable exc)	About to throw the exception *exc*
Throwing(String subloc, Throwable exc)	Same as above but with specific method name

- assertion: to test a condition and output an error message, normally with the assertion included (“Assertion failed: <assertion test>”) when the evaluation is false

Method	***Description***
assertion(Boolean assertion, String desc)	Evaluate the assertion, if false, print *desc* with the default message: “Assertion failed: <*desc*>” where <*desc*> is the assertion test itself, e.g. 5 > 3
assertion(String subloc, Boolean assertion, String desc)	Same as above but with specific method name

To reiterate, refer to the sample code with method *announce(Object o)*:

```
    public void announce(Object o) {
      String method = "announce(java.lang.Object)";
      loc.assertion(method, 5<3,  "Stupid comparison");
      try {
      }
      catch (Exception e) {
        loc.throwing(method, e);
      }
    }
```

Potential output, assuming the simplest case with ConsoleLog and default TraceFormatter:

```
    May 3, 2001 6:54:18 PM com.sap.fooPackage.Node.announce [main]
                          Error: Assertion failed: Stupid comparison
    May 3, 2001 6:54:18 PM com.sap.fooPackage.Node.announce [main]
                          Warning: Throwing java.io.FileNotFoundException:
                          C:\Not_Exist\zzzzz.log (The system cannot find the path specified)
```

4.2.4.3 Master gate

- LogT

All the other output methods of first type (with severity) are ultimately routed through this method to actually perform any logging.

Location: basically, these are the same as the first type of method, `xxxxxT( )`, but only with an additional severity argument at the beginning:

```
logT(int severity, String message)
logT(int severity, String subloc, String message)
logT(int severity, String message, Object[] args)
logT(int severity, String subloc, String message, Object[] args)
```

Category: (same situation as *Location):*

```
logT(int severity, Location loc, String message)
logT(int severity, Location loc, String subloc, String message)
logT(int severity, Location loc, String message, Object[] args)
logT(int severity, Location loc, String subloc, String message,
      Object[] args)
```

4.3 Typical Practice

The steps and logic of doing logging should be fairly clear by now.
Certainly, the information given is adequate to get users started on instrumenting logging in their code. However, a good style of doing logging is recommended:
Enabling logging in two major steps: initialization and the individual source classes.

Two stages : higher level vs individual level
Applications are normally made up of different components and are deployed in a distributed environment. Typically, there should exist a common initialization routine for a component. Logging configuration, such as setting severity, assigning destination logs, can be done at this stage as well. Then, the actual insertion of trace/log messages will be done in the respective Java class level at the reasonable points.

A top-down approach is used, that is, defining coarse configuration at a higher level (component-oriented parent node). Most of the time, users do not need to fine-tune the logging behavior in such details for each class. Then, a basic configuration at the top level will be good enough to get logging started. Only when specific setting has to be done individually for certain classes, then the users can as well enable that in the class level.

Component level at initialization
For example, a component 'ComX' under SAP may have the following package hierarchy: *com.sap.comX.xxx.xxxx.xxx*. During initialization, if logging is desired for monitoring important messages (with severity error or above) to be shown at the console, one should set up something like this:

```
Location _loc = Location.getLocation("com.sap.comX");
_loc.setEffectiveSeverity(Severity.ERROR);
_loc.addLog(new ConsoleLog());
```

Class level
In the individual class, it is recommended to have a static variable to hold the access to the location object (for better performance) and then start inserting messages:

```
Static Location _loc = Location.getLocation(<classname>.class);
……
_loc.infoT(……….);
_loc.errorT(……….);
```

The code will be cleaner with this 2-level setting. And the reason why this works is because of the hierarchical feature implemented in the logging framework. Refer to Section 4.4.2 and 4.4.3 for more details.

You will gradually find that the actual logging part within each class is pretty static, only the configuration part to manipulate the logging behavior is dynamic, such as, output volume, output destination, output format…. This will be covered in details in Section 4.6.

These two sections provide the basic concepts and show you the basic steps to get you started. In order to benefit the most out of the tool, you should explore around with the references of the next few sections.

4.4 Basic Features

This section describes a few tricks that enhance the intelligence to meet the common logging requirements.

4.4.1 More configuration options

Refer to **Figure 2-1**, it shows the options to alter the logging output through the use of formatter, log and filter. These are the fundamental means to control how the message should be presented, where it should be written to or whether it should be screened out.

4.4.1.1 Log (destination)

A *log* represents the destination where the messages to be written to. It is also the object we will specify and assign an appropriate *formatter* for the *log*, as well as multiple optional *filters* (see the next two sections). Like the *log controller* source object, a severity level can be defined for each *log*, but unlike the default for *log controller* (*Severity.NONE)*, the default for a *log* is *Severity.ALL*, that is, no effect on the main severity level check with *log controller*. (See **Appendix C** for an overview of default values.)

Currently, the following logs are provided:

- **ConsoleLog**
  - Direct the messages to *System.err* (`java.lang.System`)
  - Typical use in debugging process for a quick view of problems
- **FileLog**
  - Direct the messages to a file or a set of rotating files (more details in Section 4.4.5)
- **StreamLog**
  - Direct the messages into an arbitrary *OutputStream* (`java.io.OutputStream`)

4.4.1.2 Formatter

Each type of log destination can print the messages in different formats. Currently, there are three major types of formatter:

- **TraceFormatter**
  - Human readable format
  - This is very likely to be the most commonly used formatter when users want to quickly understand what is going on with the application.
  - Normally, not used with log viewers (refer to Section 6.2). Therefore, users can customize the pattern of the TraceFormatter, indicated by various placeholders. Details can be referred to the Javadoc of method '`setPattern`'. By default, the pattern is:

    ```
    %24d %-40l [%t] %s: %m
    ```

    and the corresponding output is:

    ```
    Jan 01, 2001 10:10:00 PM      com.sap.FooClass.fooMethod [main] Fatal: A sample fatal
    message
    ```

    For example, if the pattern becomes:

    ```
    %24d %l [%s] %m
    ```

    The corresponding output is:

    ```
    Jan 01, 2001 10:10:00 PM com.sap.FooClass.fooMethod [Fatal] A sample fatal message
    ```

  - The number in the pattern denotes the width of a field. In the first example: "%-40l" indicates a 40 char limit for the location name (com.sap.FooClass.fooMethod) that will be right aligned (with a minus sign). If the length of a string exceeds the width defined, the string will be truncated. The second example: "%l" simply displays the full string that will be left aligned.
  - The meaning of the place holders (details can be found in Javadoc):

Placeholder	*Description*
%d	timestamp in readable form

%l	the location of origin (eg: com.sap.FooClass.fooMethod)
%c	the log controller the message was issued through (eg: com.sap.FooClass)
%t	the thread that emitted the message
%s	the message severity
%m	the formatted message text
%I	the message id
%p	the time stamp in milliseconds since January 1, 1970 00:00:00 GMT
%g	the group identification

Table 4-1 Placeholders of TraceFormatter

- **XMLFormatter**
  - Suitable for file transfer to be further processed in other applications
  - Sample out of the same result is:

```
<record>
  <id>10.48.27.165:4A5AB2:E99D2EDAFF:-8000</id>
  <time>Mon Jan 01 22:00:00 PDT 2001</time>
  <source>com.sap.FooClass</source>
  <location>com.sap.FooClass.fooMethod</location>
  <thread>main</thread>
  <severity>Fatal</severity>
  <msg-type>Plain</msg-type>
  <msg-clear>A sample fatal message</msg-clear>
</record>
```

  - The DTD of the result is:

```
<!ELEMENT table (id, time, source, location, thread, group?, severity, relatives?, msg-
                type, msg-code?, bundle?, msg-clear?, args?)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT time (#PCDATA)>
<!ELEMENT source (#PCDATA)>
<!ELEMENT location (#PCDATA)>
<!ELEMENT thread (#PCDATA)>
<!ELEMENT group (id, level, indent)>
<!ELEMENT level (#PCDATA)>
<!ELEMENT indent (#PCDATA)>
<!ELEMENT severity (#PCDATA)>
<!ELEMENT relatives (relative+)>
<!ELEMENT relative (#PCDATA)>
<!ELEMENT msg-type (#PCDATA)>
<!ELEMENT msg-code (#PCDATA)>
<!ELEMENT bundle (#PCDATA)>
<!ELEMENT msg-clear (#PCDATA)>
<!ELEMENT args (arg+)>
<!ELEMENT arg (#PCDATA)>
```

- **ListFormatter**
  - Output in this format serves like a router to send the data to be processed further by another application, e.g. a log viewer, instead of being read directly by an end-user
  - As simple hash-separated fields to be processed mainly with a log viewer
  - Sample output of the same result is, as of version 1.3:

```
#1.3#10.48.27.165:4A5AB2:E99D42D4F4:-8000#Mon Jan 01 22:00:00 PDT
2001#com.sap.FooClass#com.sap.FooClass.fooMethod#main##0#0#Fatal##Plain###A
sample fatal message#
```

- The order of the fields, delimited by the hash sign is:
    - Version (of the ListFormatter)
    - Message id
    - Timestamp
    - Source name (log controller)
    - Location name (the actual code location that generates the message)
    - Thread id
    - (Group id)
    - Group level
    - Group indentation
    - Message severity
    - (Relatives names: can be multiple)
    - Message Type (PLAIN or JAVA)
    - (Message Code)
    - (ResourceBundle name)
    - Message
    - (Number of arguments)
    - (Arguments)

4.4.1.3 Filter

Refer to **Figure 2-1**, filter can be added to both *LogController(source)* and/or *Log(destination)* to further restrict or alter the output behavior.
Multiple filters are allowed to assign to each *LogController* and/or each *Log*.

<u>Implementation by users</u>
Actual implementation of filter has to be done by users, according to the interface defined in the package. Basically, this evaluates the *LogRecord* object and veto accordingly. The name of the only method indicates that, a boolean *true* is returned when the *logRecord* passes the filter, otherwise *false*.

Since the implementation is done by users, classname of the filter will not be known in advance. If the user-defined filter class is to be specified in the configuration file, a fully qualified classname should be used. Be careful that this only works if the class is already included in the classpath during the configuration. Should it needs its own classloader, the latter has to be set clearly in the configurator constructor. Details of configuration tool can be found in Section 4.6.

4.4.2 Hierarchical severity inheritance

By now, you should be familiar with the severity scale defined in this tool. In Section 4.2.2, you learn how to assign severity to a source (*location* or *category*). Normally, there are numerous source objects in your program, and the task will become very tedious if severity has to be assigned individually to each of them. Thus, the requirement of a hierarchical naming convention (as mentioned in Section 4.2.1) has its reason now.
<u>Interfere inheritance</u>
Intuitively, severity assigned to the parent node will be effective for its children and all descendants. By and large, this is true. But there is little more that allows you to control the inheritance behavior. Descendants do not need to inherit the severity from parent with the following **three ways**:

- **Explicit severity setting** (default: *Severity.NONE*)

As long as the child object has been explicitly assigned a severity, this will win, and will not be affected by the severity assignment of its ascendants. This corresponds to the API:

```
static final Location loc = Location.getLocation(com.sap.foo);
loc.setEffectiveSeverity(Severity.INFO);
```

So, if parent '`com.sap`' is assigned '*Severity.FATAL*', this has no influence on '`com.sap.foo`', the latter still has '*Severity.INFO*' assigned.

By the same token, you can easily deduce that descendants (without explicit severity assignment) will inherit the severity from the **closest ascendants** that has explicit *effective* severity assigned. With the same example shown above, what will be the severity of '`com.sap.foo.child1`'? *FATAL* or *INFO*?
Remember, all descendants of '`com.sap.foo`' will now inherit severity from '`com.sap.foo`'.

- Define a severity floor limit: **Minimum severity** (default: *Severity.ALL*)
  Assigned severity should not be more relaxed than this
- Define a severity ceiling limit: **Maximum severity** (default: *Severity.NONE*)
  Assigned severity should not be stricter than this

You can specify the floor and ceiling thresholds individually or together. These two define the valid severity range for a source object. Therefore, by default, it does not restrict any inherited severity because it must fall within the default minimum severity *ALL* and maximum severity *NONE*.

<u>Example</u>
Consider the following, where explicit severity has NOT been set for '`com.sap.foo`', but the range has been defined as:

```
static final Location loc = Location.getLocation(com.sap.foo);
loc.setMinimumSeverity(Severity.DEBUG);
loc.setMaximumSeverity(Severity.WARNING);
```

When parent '`com.sap`' is explicitly assigned with '*Severity.FATAL*', this severity will not be inherited straightly by '`com.sap.foo`' because it gets denied by its ceiling threshold. Instead, '`com.sapmarkets.foo`' will get the ceiling threshold '*Severity.WARNING*'. The same theory applies to the floor threshold, i.e., any severity of parent that is lower than '*Severity.DEBUG*' will be ignored.
Only when the parent's severity falls between the severity range of the child can overwrite the child's severity with its own value.
(A technical side note: assigning explicit severity, `setEffectiveSeverity(Severity.INFO)`, automatically overwrites the value of both *MinimumSeverity* and *Maximum Severity* into *Severity.INFO*).

Let's try to organize the possibilities here:

A default setting:

Location/ Category	*Maximum Severity*	*Minimum Severity*	*Resulting (Effective) Severity*	*Description*
com.sapmarkets	NONE	ALL	NONE	default
com.sap.foo1	NONE	ALL	NONE	default
com.sap.foo1.child1	NONE	ALL	NONE	default
com.sap.foo2	NONE	ALL	NONE	default
com.sap.foo2.child1	NONE	ALL	NONE	default

Scenario 1:

Location/ Category	*Maximum Severity*	*Minimum Severity*	*Resulting (Effective) Severity*	*Description*
com.sap	WARNING	WARNING	WARNING	setEffectiveSeverity = WARNING
com.sap.foo1	FATAL	ALL	WARNING	setMaximumSeverity = FATAL
com.sap.foo1.child1	NONE	ALL	WARNING	
com.sap.foo2	NONE	ALL	WARNING	
com.sap.foo2.child1	INFO	ALL	INFO	setMaximumSeverity = INFO

Scenario 2:

Location/ Category	*Maximum Severity*	*Minimum Severity*	*Resulting (Effective) Severity*	*Description*
com.sap	INFO	INFO	INFO	setEffectiveSeverity = WARNING
com.sap.foo1	FATAL	FATAL	FATAL	setEffetiveSeverity = FATAL
com.sap.foo1.child1	NONE	ALL	FATAL	
com.sap.foo2	NONE	ALL	INFO	
com.sap.foo2.child1	FATAL	WARNING	WARNING	setMinimumSeverity = WARNING

4.4.3 Hierarchical destination inheritance

The naming hierarchy does not only enable inheritance severity, but also destination (*log)* assignment. The logic is even more straightforward here:

```
static final Location parent = Location.getLocation(“com.sap”),
             Location child =
                           Location.getLocation(“com.sap.foo”);
parent.addLog(new ConsoleLog());
parent.fatalT(“A fatal message from parent”);
child.fatalT(“A fatal message from children”);
```

Both messages will be output to the console, even *ConsoleLog* is assigned only to the parent node.

Additivity

Multiple destination logs are allowed. So, unlike the severity inheritance logic, the assignment of logs to a source object is additive. There is no race condition. For example, adding to the previous code (original code greyed out):

```
static final Location parent = Location.getLocation("com.sap"),
             Location child =
                          Location.getLocation("com.sap.foo");
parent.addLog(new ConsoleLog());
parent.fatalT("A fatal message from parent");
child.fatalT("A fatal message from children");
child.addLog(new FileLog(<a file>));
```

In the Console, there will still be two messages printed, each from parent and child, while in the output file <a file>, messages from child (here, only 1 message) are also printed, in addition to Console destination.

One might want to suppress this inheritance feature in certain circumstances to fine-tune the output behavior. This is doable, as you can find more details in Section 4.5.1.

4.4.4 Severity evaluation

Whether a message can be logged or not, largely depends on its severity level compared against the severity threshold assigned to the source object (another factor is the use of filter, if any). In some cases, developers do not feel comfortable in simply calling the output API (logT, fatalT, errorT, …) due to potential performance issue, for example, resolving the parameter list of the call. An additional severity check before making the output calls is desired.

There are APIs allow you to do this. You can find them under the class *LogController*. They correspond to the output APIs and they are:

```
beLogged(Severity level)
beFatal(), beError(), beWarning(), beInfo(), bePath(), beDebug()
```

These return a boolean *true* if the message passes the respective severity level check, otherwise *false.*

4.4.5 Output File

Directing messages into a file is a very popular practice. In this tool, this is done by assigning *FileLog* to your source objects. In fact, the Javadocs has very clear and detailed instruction on this. This section will recap and explain certain common configuration features to complement the Javadocs.
Refer to **Appendix C** for the summary of default behavior. We have mentioned the *FileLog* class twice in the two tables. By default, messages written to an output file will be in the *ListFormatter* format, without using any special character encoding. And there will only be one single output file, increasing in size constantly. There are number of options that you can configure the behavior of an output file, via API or configuration file (refer to Section 4.6).

- Filename
- Limit file size and do sequencing on the output file
- FileLog vs Physical file

4.4.5.1 Filename

Filename can be expressed as the fully qualified filepath, such as “C:\temp\trace.log”, or expressed in pattern with a number of available placeholders. This can take care of the potential hardcoding problem and platform dependent issue, such as filename separator.

Place holder	*Description*	*Comments*
/	Local file name separator	“C:\temp\trace.log” == “C:/temp/trace.log” but probably you would like to use the latter
%h	Home directory	Value of system property: ‘*user.home*’
%t	System temporary directory	Value of system property: ‘*java.io.tmpdir*’. Normally, it is “C:\temp”. Therefore: “C:\temp\trace.log” == “%t/trace.log”
%u	Unique number to make the file name unique	GUID. To make sure no naming conflict will occur.
%%	The percentage sign	
%g	Sequence number of the file	This is useful only when users specify the file size limit and the max. number count to do rotation. By default, this will be appended to the end of the filename/filepattern specified. E.g. “%t/trace.%g.log” ----> “C:\temp\trace.0.log” (otherwise, “C:\temp\trace.log.0” by default)

Table 4-2 Placeholders of filename

4.4.5.2 Output file sequence and rotation

This is implicitly done when users have specified values for the filesize (in byte) and the cap for the sequencing count before rotating back to the first file. This pair of parameters has to **coexist** together. Either they both have values assigned or both values equal to zero. Otherwise, depends the option you are using to do this configuration, either an exception can be thrown (through programming API) or values with best guess will be used (through configuration file, see Section 4.6), e.g. both attributes get default zero values (refer to Section 5.1 for this fault-tolerance logic).

This can be done when calling the *FileLog* constructor:

```
FileLog     _file = new FileLog(“%t/trace.%g.log”,
                                800000,
                                10,
                                new TraceFormatter());
```

Or a quick peek on the configuration file syntax (refer to Section 4.6):

```
log[File]                          = FileLog
log[File].pattern                  = %t/trace.%g.log
log[File].limit                    = 800000
log[File].cnt                      = 10
log[File].formatter                = TraceFormatter
```

The first output file will be created “C:\temp\trace.0.log” (NOTE: sequence number starts with 0), and when its size approaches and then exceeds the limit (800,000 bytes), the next incoming message will be directed to a new file “C:\temp\trace.1.log”. This process keeps going until the last sequenced file “C:\temp\trace.9.log” (count = 10), then the next file to be written will again become “C:\temp\trace.0.log”.

4.4.5.3 FileLog vs physical file

A *FileLog* is a logical representation for a physical output file where messages are directed to. With the several configuration options mentioned in the previous section, users can manipulate the output behavior of the output file through the API of *FileLog*, as well as do (multiple) assignment(s) of the *FileLog* to (various) source object(s).

1:1 recommended
Intuitively and normally, this should be a 1:1 relationship between the *FileLog* and the actual output file. It is rare that a user will want to create two instances of *FileLog* on the same file and with different configuration, e.g. one with *TraceFormatter* and the other one with *XMLFormatter*. Currently, this is not illegal. But it does not look like a reasonable design, and user will run into the risk of improper synchronization of messages in a multi-threading environment. The full message text may get interrupted and interweaved with another message text.

Therefore, this is **NOT** recommended. At the meantime, please pay attention to this when coding or doing configuration.

4.4.6 Change Output Logs/Filters

The decoupled design of the logging framework makes it flexible to configure the logging behavior. One example is to change the log destination assignment to a source object. As mentioned in Section 4.4.3, the assignment of *logs* to the source objects is additive. This applies to filter assignment too.

That is, adding an additional output destination is straightforward, but if you need to switch the output destination, e.g. from a console to a file, user must include instructions to remove the *ConsoleLog* from the source object and then assign a *FileLog* to it. The APIs that support this:

```
        _loc.removeLog(<a specific log>);
then,   _loc.addLog(<the new log>);
```

For filters, the APIs are similar:

```
        _loc.removeFilter(<a specific filter>);
then,   _loc.addFilter(<the new filter>);
```

Eventually, update of this kind is very likely to be done with configuration file (Section 4.6), and changes to logging properties during runtime are supported. There is a special syntax to deal with additive attributes. Refer to the end of section 4.6.2.3. A quick peek again:

```
log[File]                          = FileLog
log[File].pattern                  = %t/trace.log
log[File].formatter                = ListFormatter

com.sap.logs                  = log[File]

## (1) in the next round, user could switch the output to Console:
com.sap.logs                  = ConsoleLog
## (2) or you can add additional console ouput:
com.sap.logs                  = + ConsoleLog
```

4.4.7 Language Dependency

Mainly for log messages
This is rarely an issue for *trace messages* that mainly used by developers during the debugging process. On the other hand, *log messages*, which are related to specific logical areas, are most likely to be read by administrators, end-user support group, consulting group… to diagnose application problem during runtime. Not only a meaningful text should be logged, but also a message translation mechanism should be supported as application may be deployed worldwide.

ResourceBundle

This logging tool does support language dependency with the use of *ResourceBundle* provided in Java. (For more background about this, please refer to the java class *java.util.ResourceBundle*). The basic concept is, the message text logged by the developers in their program is 'neutral'. It is like a message code/id and a resource bundle file for each language exists to map the message code into the text of the respective language. During the application runtime, depends on the locale, resource bundle file will be looked up, and message text will be translated and localized accordingly.

More APIs

To meet this requirement, there exists a few more output methods API under *Category* class to enable the translation feature for the *log messages*. These do not appear under the *Location* class (in terms of the basic ones you have learnt in Section 4.2.4.1 that do not involve the reference to *category*).

Recall the output methods of *Category* you have learnt in Section 4.2.4.1 and 4.2.4.3. If you are familiar with them, you should have no problem in using these additional output methods to enable translation, because they are very similar and have the same overloading pattern. Basically, these are methods

- having identical corresponding method names, but ended without a '*T*'
- replacing the argument `'String message'` into `'Object msgCode'`

The table below shows a few commonly used methods, compared to the 'classic' ones that you have learnt previously.

Category ('xxxx' stands for the severity level)	***Translation supported***
xxxxT(Location loc, String message)	xxxx(Location loc, Object **msgCode**)
xxxxT(Location loc, String subloc, String message)	xxxx(Location loc, String subloc, Object **msgCode**)
xxxxT(Location loc, String message, Object[] args)	xxxx(Location loc, Object **msgCode**, Object[] args)
xxxxT(Location loc, String subloc, String message, Object[] args)	xxxx(Location loc, String subloc, Object **msgCode**, Object[] args)
logT(int severity, Location loc, String message)	log(int severity, Location loc, Object **msgCode**)
logT(int severity, Location loc, String subloc, String message)	log(int severity, Location loc, String subloc, Object **msgCode**)
logT(int severity, Location loc, String message, Object[] args)	log(int severity, Location loc, Object **msgCode**, Object[] args)
logT(int severity, Location loc, String subloc, String message, Object[] args)	log(int severity, Location loc, String subloc, Object **msgCode**, Object[] args)

Table 4-3 Common Output API with support of translation

Sample

MsgCode is the 'message id' specified in the resource bundle.
For example, there are two resource files; we have the following translation (just to make fun of British English and American English…). *MsgCode* is the key specified in the left of the pair.
(1)
```
DBProblem          = Problem with Database.
ProcessCanceled    = Process {0} canceled.
WrongColor         = Wrong Color: {0}.
```
(2)
```
DBProblem          = Problem with Database.
```

```
ProcessCanceled    = Process {0} cancelled.
WrongColor         = Wrong Colour: {0}.
```

In the program, the code becomes:

```
        _cat.error(_loc, "ProcessCanceled");
```

During runtime, depends on the specified locale (or the default locale of the platform), the output will be either:

```
        Process #### canceled.
```

Or

```
        Process #### cancelled.
```

Invalid translation

What if the *MsgCode* specified is invalid? A `MissingResourceException` will be thrown. In this case, the exception will be caught and logged by the logging framework itself without jeopardizing the flow of the application. The value string of *MsgCode* itself will be used as the translated string. This is the best substitute in this situation. (This is actually the 'fault-tolerance' mechanism provided by the logging tool. See Section 5.1 for details).

There exists another option that developer can specify a meaningful message string as a **backup** to get around with the invalid *MsgCode* problem. Then, the logging framework needs not guess a substitute value, and this backup string can give more meaningful message (although non-translated) than a *MsgCode*. This is supported by the use of an additional argument *MsgClear*. Note that the same set of method APIs is overloaded again with this additional argument (method overloading is actually very systematic, so, don't get daunted by the number of methods).

Fallback supported
xxxx(Location loc, Object msgCode, String **msgClear**)
xxxx(Location loc, String subloc, Object msgCode, String **msgClear**)
xxxx(Location loc, Object msgCode, Object[] args, String **msgClear**)
xxxx(Location loc, String subloc, Object msgCode, Object[] args, String **msgClear**)
Log(int severity, Location loc, Object msgCode, String **msgClear**)
Log(int severity, Location loc, String subloc, Object msgCode, String **msgClear**)
Log(int severity, Location loc, Object msgCode, Object[] args, String **msgClear**)
Log(int severity, Location loc, String subloc, Object msgCode, Object[] args, String **msgClear**)

This is a fallback option when the use of *MsgCode* is not very reliable value. When argument *MsgClear* (non-null) is available, its value **always** exists in parallel with the *MsgCode*. This option becomes particularly useful when log viewers are involved. In most cases, the assumption is *MsgCode* will be used by log viewer and translation is done at the time of viewing whenever the *MsgCode* is valid, otherwise, *MsgClear* can be used as a backup.

4.5 Advanced Features

4.5.1 Three forms of log assignment

Normally, users like to take advantage of the inheritance property to simplify the configuration task, e.g. for both severity and log assignment. The assignment of log is additive, and at times, more control over the log assignment is needed to fine-tune the specific assignment for certain source objects at certain levels.
There actually exist three kinds of log which you can assign to a source object:

- ‘common’ log
- local log
- private log

They are **mutually exclusive** for each source object. The assignment of one type will automatically disable the assignment of the other type (if any).

4.5.1.1 ‘Common’ Log

So far, this is the one that has been introduced previously. This allows regular inheritance; this log will become available to all descendants of the parent object.
The corresponding API is:

```
static final Location parent =
                        Location.getLocation(“com.sap”),
             Location child =
                        Location.getLocation(“com.sap.foo”);
parent.addLog(new ConsoleLog());
parent.setEffectiveSeverity(Severity.NONE);
child.setEffectiveSeverity(Severity.INFO);
child.fatalT(“A fatal message from children”);
```

This is an unconditional inheritance. As long as the message passes the evaluation of severity and filter of the *child* object, it will be printed out via the inherited log, in this case, ConsoleLog.

4.5.1.2 Local Log

Inheritance is allowed, but with additional condition: final discretion of message printing lies with the original *parent* object. The log is local, in the sense that it is not available for the descendants if the *child* message does not pass the severity and filter test of the *parent* object.

```
static final Location parent =
                        Location.getLocation(“com.sap”),
             Location child =
                        Location.getLocation(“com.sap.foo”);
parent.addLocalLog(new ConsoleLog());
parent.setEffectiveSeverity(Severity.NONE);
child.setEffectiveSeverity(Severity.INFO);
child.fatalT(“A fatal message from children”);
```

Contrary to the previous example, the child message will not be printed because it does not pass the severity test of the parent.

4.5.1.3 Private Log

This completely disables inheritance. Log assignment is private: only effective for the parent source object.

```
static final Location parent =
                        Location.getLocation(“com.sap”),
             Location child =
                        Location.getLocation(“com.sap.foo”);
parent.addPrivateLog(new ConsoleLog());
```

```
child.setEffectiveSeverity(Severity.INFO);
child.fatalT("A fatal message from children");
```

In this case, child object does not inherit any destination log at all. Nothing will be printed even the message passes all the severity and filter tests.

4.5.2 Relations between Category & Location

(This section and the next are closely related.)

Reason

It is a common practice to look at the log messages and trace messages together when doing diagnosis. A correlation between a problematic logical area and the source code location that generates the problem is highly desired. For example, an error occurs when closing the database, the actual location of the source code (from which class, which method, with what argument(s)) is reported as well.

You have already seen the introduction of how this can be supported in Section 4.2.4.1 to generate both log and trace messages in parallel, with the use of *category* and *location*.
We will elaborate an example here.

Example

For simplicity, we only show the example of attaching a *location* to a *category* (we will here refer to the *location* being associated to the category as **relative** of the category), but this is also true for the vice versa case.
Note that the API supports precisely the assignment of a *category* to a *location* and also the other way round, but **not** among the same type. It is legitimate to associate more than one *category* to a *location* at each logging, but only one *location* for one *category*. Refer to the output method APIs of each class.

```
Package com.sap.fooPackage;

import com.sap.tc.logging.*;

public class Node {
  private static final Location loc =
            Location.getLocation("com.sap.fooPackage.Node");
  private static final Category objMgmt =
            Category.getCategory("/Objects/Management");

  public void announce(Object o) {
    final String method = "announce(java.lang.Object)";
    loc.entering(method, new Object[] {o});
    try {
      // Register object ...
    }
    catch (RegistrationException e) {
      objMgmt.errorT(loc,
                     method,
                     "Error registering object {0}.",
                     new Object[] {o});
    }
```

```
      loc.exiting();
    }  // method announce
  }  // class Node
```

In order to output all the trace and log messages highlighted in the example above, user will have the following severity setting, for example:

```
        loc.setEffectiveSeverity(Severity.PATH);
        objMgmt.setEffectiveSeverity(Severity.ERROR);

        conLog = new ConsoleLog();
        loc.addLog(conLog);
        objMgmt.addLog(conLog);
```

Output
And for the output line from the *category* '`objMgmt`', it will output 2 messages simultaneously: 1 log message and 1 trace message. They will have the same message id for **cross-referencing** each other. This makes the analysis more comfortable.
(But recall the information in sections 4.3, 4.4.2 and 4.4.3, the configuration could have already been taken care of in the initialization stage and is effective to this local scope through the hierarchical inheritance logic).
If the *location* has stricter severity setting, e.g. default *Severity.NONE*, all the trace output will be suppressed, including the one from the *category*, that is, that output line will NOT produce two messages simultaneously, but only the log message.

More control
Naturally, the next question may be whether we can do more advanced configuration regarding the correlated *category* and *location* source objects. This is positive.
Consider *Category* "/Objects/Management"; we only want to focus on some extreme situations, that is, messages with severity *FATAL*. Several source code *location* ('`com.sap.xxx.a`', '`com.sap.xxx.b`', ….) can result in a fatal condition in this logical area. And for some reasons, we are particularly interested in one of them, e.g. '`com.sap.xxx.a`' and would like to generate more output messages, including all with severity *INFO* or above related with this *location* only, while maintaining *FATAL* for the rest.

Having a tight severity control initially and then relaxing it for particular areas in question to produce more output for analysis is essential. More details will be shown in the next section, which shows how to achieve this by playing with the 'relative severities'.

4.5.3 Relative severities

In the previous section, we showed that you could tag along a log controller to another log controller. In this case, the amount of output can be configured with respect to the relative(s) too. The evaluation of the message output not only depends on the log controller's own severity level, but also the relative severity of its relative(s), if any.

4.5.3.1 Set up relative severity (add relative)

The API is really similar to the 'regular' one:

Regular	***Location***	***Category***	***Comment***
setEffectiveSeverity (int severity)	setEffectiveSeverity (Category relative, int severity)	setEffectiveSeverity (Location relative, int severity)	
setEffectiveSeverity()	setEffectiveSeverity (Category relative)	setEffectiveSeverity (Location relative)	Reset severity (with respect to the relative)
setMaximumSeverity (int severity)	setMaximumSeverity (Category relative, int severity)	setMaximumSeverity (Location relative, int severity)	
setMaximumSeverity()	setMaximumSeverity (Category relative)	setMaximumSeverity (Location relative)	Reset
setMinimumSeverity (int severity)	setMinimumSeverity (Category relative, int severity)	setMinimumSeverity (Location relative, int severity)	
setMinimumSeverity()	setMinimumSeverity (Category relative)	setMinimumSeverity (Location relative)	Reset

Table 4-4 Setting relative severity

Normally, you still assign severity in the regular fashion, as those listed in column 1. The typical requirement is to assign a stricter severity in the primary source object, and a more relaxed level tagged with a certain relative (not the other way round!).
The relative severity is specific to the pair of the location and category. When you want to change the relative severity or reset it, the relative has to be clearly indicated in the API.
The relative severity is **unidirectional**:

```
loc1.setEffectiveSeverity(cat1, Severity.INFO);
```

The following is not implicitly done:

```
cat1.setEffecitiveSeverity(loc1, Severity.INFO);
```

Relative severity is also **hereditary**. Descendants also inherit the relative severity assignment. The same logic of local restriction through the use of maximum(ceiling) and minimum(floor) severity limit (described in Section 4.4.2) is also applicable to the descendants. Note that evaluation is always done with respective to the relative(s).

4.5.3.2 Evaluation of severity with relatives

The severity of a log controller can be different when a relative severity (or severities) has been set. The rule of thumb is: inclined to a more relaxed setting.
That is, the **minimum** severity of the two factors below will always win:

- direct severity of itself
- severity related to its relative
    - if there exists multiple relatives, the minimum of the relative severities wins

This applies to all 3 types of severities shown in the table above: *effective, maximum, and minimum.*The following tables highlights the severity results in both cases of evaluating with and without a relative.

Regular: setEffectiveSeverity (int severity)	***With Relative: setEffectiveSeverity (<log controller> relative, int severity)***	***Result: getEffectiveSeverity()***	***Result: getEffectiveSeverity (LogController relative)***	***Comments***
FATAL	<…not set…>	FATAL	FATAL	
<…not set…>	INFO	NONE	INFO	
FATAL	INFO	FATAL	INFO	
INFO	ERROR	INFO	INFO	Not stricter

Regular: setMaximumSeverity (int severity)	***With Relative: setMaximumSeverity (<log controller> relative, int severity)***	***Result: getMaximumSeverity()***	***Result: getMaximumSeverity (LogController relative)***	***Comments***
FATAL	<…nothing…>	FATAL	FATAL	
<…not set…>	INFO	NONE	INFO	
FATAL	INFO	FATAL	INFO	
INFO	ERROR	INFO	INFO	Not stricter

Regular: setMinimumSeverity (int severity)	***With Relative: setMinimumSeverity (<log controller> relative, int severity)***	***Result: getMinimumSeverity()***	***Result: getMinimumSeverity (LogController relative)***	***Comments***
FATAL	<…nothing…>	FATAL	FATAL	
<…not set…>	INFO	NONE	INFO	
FATAL	INFO	FATAL	INFO	
INFO	ERROR	INFO	INFO	Not stricter

As you can see, the use of relative severity is pretty independent. It never affects the individual severity evaluation of the log controller itself. If users decide not to configure the output with respective to any relative, the relative severity assignment has no effect on the output. The result will stay intact as listed in column 3.

4.5.3.3 Output resulting from relative severity

Use the table below to elaborate (*effectiveSeverity)*. What will be the result of the following code for each of the 4 conditions listed? This is an example to produce log messages with respect to a *location:*

```
   Category cat = Category.getCategory(“Objects/Management”);
   Location loc = Location.getLocation(“com.sap.xxx.a”);
   cat.warningT(loc,
                methodname,
                "Error registering object {0}.");
```

Regular: cat.setEffectiveSeverity (<severity>)	***With Relative: cat.setEffectiveSeverity (loc, <severity>)***	***Result, output enabled (with relative ‘loc’)? cat.warningT(loc, ….)***
FATAL	<…not set…>	NO
<…not set…>	INFO	YES
FATAL	INFO	YES
INFO	ERROR	YES

Another example: to produce traces from *location* with respect to a *category*:

```
   Category cat  = Category.getCategory(“Objects/Management”);
   Location loc = Location.getLocation(“com.sap.xxx.a”);

   loc.warningT(methodname, messageCode);
   loc.warning(cat, methodname, messageCode);
```

Regular: loc.setEffectiveSeverity (<severity>)	***With Relative: loc.setEffectiveSeverity (cat, <severity>)***	***Result, output enabled (without relative ‘cat’)? loc.warningT(………..)***	***Result, output enabled (with relative ‘cat’)? loc.warning(cat, ……)***
FATAL	<…not set…>	NO	NO
<…not set…>	INFO	NO	YES

FATAL	INFO	NO	YES
INFO	ERROR	YES	YES

When the output methods API denote a relative(s), the logging framework tries to output the message **simultaneously** to both the log controller and its relative(s). The evaluations are **independent** and done separately. The results shown in the tables above are those for the log controller itself.
Whether the output is enabled at the relative totally depends on the configuration set by the relative itself, which at the same time, evaluate the severity with respective back to the log controller. If no explicit relative severity is configured by the relative, the situation becomes similar to the conditions listed in the table below. (Log a message from a category with a relative location).

Regular: cat.setEffectiveSeverity (<severity>)	*With Relative: cat.setEffectiveSeverity (loc, <severity>)*	*Output enabled for cat? cat.warningT(loc, ….)*	*Output enabled by relative 'loc'? (no explicit severity setting)*
FATAL	<…not set…>	NO	NO
<…not set…>	INFO	YES	NO
FATAL	INFO	YES	NO
INFO	ERROR	YES	NO

For the case that relative severity is also specified at the relative *loc* (assuming its own severity remains the default *Severity.NONE* value).

Regular: cat.setEffectiveSeverity (<severity>)	*With Relative: cat.setEffectiveSeverity (loc, <severity>)*	*At loc: loc.setEffectiveSeverity (cat, <severity>)*	*Output enabled by relative 'loc'?*
FATAL	<…not set…>	<…not set…>	NO
<…not set…>	INFO	INFO	YES
FATAL	INFO	WARNING	YES
INFO	ERROR	ERROR	NO

4.5.3.4 Summary of behavior

- Relative severity assignment is precise with the pair, unidirectional
  - Not affecting severity of log controller itself
  - Not affecting the severity of log controller being attached
- Therefore, output methods also have to be explicit with which relative(s)
- Severity relative is hereditary
- Also, explicit relative assignment to be specified for the relative itself, if so desired
- Evaluation always results in a more relaxing condition (allows more output)
- Evaluation is done independently for the calling log controller and the relative log controller. Each evaluation treats the other log controller as relative and does the calculation with respective to its own relative severity configuration (if any).

4.5.3.5 Alternative for customer-defined severities

This logging framework is providing about 8 severity levels to control the logging output. In practice, it shows that this number of levels is adequate enough for most applications. The majority of users mainly use 3 or 4 severity levels. But indeed certain users require additional severity levels for some special logging behavior, mainly for distinguishing different applications/stages, e.g., application area X, Y; initialization stage, …

Common requirement
Defining your own severity levels sounds like a straightforward solution, that is, creating a distinguished severity level to independently control the logging output for a specific stage/area. For example, during the initialization stage, to output messages that has severity level between *Severity.WARNING* and *Severity.ERROR.* Therefore, a customized severity level, e.g. 'SEVERITY_INIT' which is stricter than *WARNING* and lower than *ERROR* is desired.
However, there are a few drawbacks with this approach:

- Order of severity levels always total
- Definition of the customized severity levels is unclear to users, e.g. support group
- For developers, it is also difficult to maintain and coordinate the customized severity levels among components and applications

Solution with SAP API
This issue can be well handled by the SAP logging API which provides support for both logging *(category)* and tracing *(location).* As you have already learnt the details, you should be aware of the flexibility to adjust severity level independently for different focuses with the use of relative severity, e.g. more trace messages for initialization stage 'initialization'. This can be achieved easily by the following line, with *category* 'initialization':

```
   loc.setEffecitiveSeverity(initialization, Severity.WARNING);
```

Then, when writing message, call with respective to this relative 'initialization':

```
   loc.errorT(initialization, <….message….>);
```

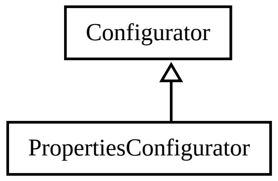
Assuming logging is not turned on for others (with default *Severity.NONE*), trace messages of severity warning or above, regarding to 'initialization' will still be written to the corresponding destination *logs*.

4.5.4 Multi-threading

This logging tool works well in a multi-threading environment.
Thread id is stored with the message, and this can be displayed or masked in the output. Recall the configuration that can be done for *TraceFormatter*, where you can specify the pattern of the format to include the thread id or not.
A sample output in *trace format*:

```
Jan 01, 2001 10:10:00 PM     com.sap.fooClass.fooMethod [Thread-0] Fatal: A fatal message
Jan 01, 2001 10:10:00 PM     com.sap.fooClass.fooMethod [Thread-1] Fatal: A fatal message
Jan 01, 2001 10:10:00 PM     com.sap.fooClass.fooMethod2 [Thread-1] Error: An error message
Jan 01, 2001 10:10:00 PM     com.sap.fooClass.fooMethod [Thread-2] Fatal: A fatal message
Jan 01, 2001 10:10:00 PM     com.sap.fooClass.fooMethod2 [Thread-0] Error: An error message
Jan 01, 2001 10:10:00 PM     com.sap.fooClass.fooMethod2 [Thread-2] Error: An error message
```

4.6 Configuration tool



This is one of the most demanded features.
A means to independently configure the behavior and output of logging, without messing with the source code is very important. The most common case is, users would want to generate more or fewer log messages, depends on the diagnosis level they are in. Sometimes, users would also want to change the output destination or even the format of the message. These all can be specified separately and dynamically integrated with the program. No recompilation of the source code is necessary.

If you are clear about the concept, when you refer back to the example shown in Section 4.2, one can immediately realize that, eventually, what are left in the source code are pretty much the API of step 1 and step 4. The rest can be taken out and placed in a properties file. The following two sections will show you how to read the properties file from your code, and how to construct a properties file.

4.6.1 To use

4.6.1.1 Calling it

Now, on top of using the API in your code to do configuration, like setting severity, designating output destination, you can now load in the configuration data from a properties file (or a Java Properties object).

For example, in your code, usually in the initialization stage:

```
   _propConfig = new PropertiesConfigurator(
                                 new File("logging.properties"));
   _propConfig.configure();
```

That is it.
The example shown above reads in a properties file named “logging.properties” in the current directory (you can fill in an absolute filepath), and triggers the configuration as specified in the file. Please refer to the Javadoc API for various PropertiesConfigurator constructors, including the option to load your own classloader.

Currently, there is not a centralized storage framework to hold the properties files. Each application should refer to its own properties management directory. Should the tool cannot locate the properties file (eg. wrong path, non-existing), an error message will be prompted, and application should continue to run, in this case, without logging being switched on (assuming all configuration is done through the properties file). Refer to Section 5.1 for details on error handling concept.

4.6.1.2 Optional periodic reloading

There exists an optional command where you can specify a periodic value (in minutes) to read and load the properties file again, to scan for any new configuration automatically. This feature will be even more useful when a *log console* is available.
The API in doing this, e.g. reloading the properties file every 30 minutes:

```
   _propConfig.setPeriodicity(30);
```

4.6.2 Syntax and semantics of properties file

The syntax rule and some semantics are clearly stated in the Javadoc API under the class “PropertiesConfigurator”. In this section, we will elaborate on the explanation and examples on the semantics in case you are not used to interpret the syntax rule. Bear in mind that this is like transforming your configuration API into a list of commands of **key-value-pair** properties represented in a properties file. So, it is a good idea to have the Javadoc API for reference while you are reading this section. This can definitely speed up your understanding with the semantics of the content.

4.6.2.1 A very simple sample content

As mentioned, it is listed in a key-value-pair format. The simplest form is:

```
<location>.attribute     = value
<category>.attribute     = value
```

(For your convenience, the attributes are bolded in the example to distinguish it from the arbitrary <location> which is also delimited by a period '.')

```
####Location "com.sap.foo"
com.sap.foo.severity         = WARNING
#### Now, assigning the log to the Location
com.sap.foo.logs              = ConsoleLog
```

Can you tell what this is doing? And can you map it back to the API?
This is equivalent to the following, where the configuration code is highlighted:

```
Location _loc = Location.getLocation("com.sap.foo");
_loc.setEffectiveSeverity(Severity.WARNING);
_loc.removeLogs();    //clean up existing attachments
                      //remember log assignment is additive?
_loc.addLog(new ConsoleLog());
_loc.fatalT(……………………..);
_loc.infoT(………………………..);
```

Now, with the use of the properties file, you can simply adjust the level of severity to control the amount of output, or even redirect the messages into another destination(s).

You should notice the following sample values in the example above:

- **Attributes** of a LogController: **severity, logs**
- The value of severity: WARNING (constant value of class *Severity*, **case-sensitive**!)
- The value of log: ConsoleLog (java class name of *ConsoleLog*, **case-sensitive**!)

Refer to the syntax rule described in the Javadoc API. Have you got a better sense of the description? Can you follow the rule and map to the sample above?

4.6.2.2 Another sample content

The previous example is very straightforward. What if you need to do more configurations, for example, assigning 2 output destinations for the location, replacing the default configuration with your own configuration on the *Log* and/or on the *Formatter*…such as assigning an *XMLFormatter* to a *ConsoleLog* (instead of using the default *TraceFormatter*). In addition, you might want to reuse your own configuration object and assign it to multiple other objects. It is no longer sufficient to use the classname *ConsoleLog* directly.
You will have to use an **identifier** (~variable) to support these configurations. The notation is not difficult: you denote the type of object, followed by the variable id in square bracket. The 2 main types are:

- log[<id>]
- formatter[<id>]

Then you can manipulate the configuration of the *log* or *formatter* object with its respective attributes, before really using it. (Refer to next section 4.6.2.4 for a summary of possible semantics.)

Imagine this is the configuration coding that we have originally. How will you represent this in your properties file?

```
Location _loc = Location.getLocation("com.sap.foo");
_loc.setEffectiveSeverity(Severity.WARNING);
TraceFormatter _trFormatter =
                      new TraceFormatter("%s: %-30l [%t]: %m");
loc.addLog(new ConsoleLog(_trFormatter));
loc.addLog(new FileLog("C:\\temp\\myTrace.txt", _trFormatter));
_loc.fatalT(………………………..);
_loc.infoT(………………………….);
```

Once you understand the following representation, you are quite ready to deal with various situations, with reference to the syntax rule indicated in the Javadoc API, in addition to the summary tables listed in the next section.

```
com.sap.foo.severity       = WARNING
#### Set up a FileLog, for storing trace, with <id>: 'File'
log[File]                         = FileLog
log[File].pattern                 = C:\\temp\\myTrace.txt
log[File].formatter               = formatter[TraceNoDate]
#### Set up a ConsoleLog, with <id>: 'Console'
log[Console]                      = ConsoleLog
log[Console].formatter            = formatter[TraceNoDate]
#### Set up a TraceFormatter, with <id>: 'TraceNoDate'
#### and its pattern starts with the Severity level, and
consists no date/timestamp
formatter[TraceNoDate]            = TraceFormatter
formatter[TraceNoDate].pattern    = %s: %-30l [%t]: %m

com.sap.foo.logs               = log[Console], log[File]
```

In the very last line, you may realize that using a ',' makes multiple entries possible.

4.6.2.3 Support of periodic reloading

There is an intelligence to deal with log assignment (additive) with periodic reloading. Technical details are not covered here but the general behavior will be explained and this has something to do with the syntax: the use of an optional '+', e.g.:

```
com.sap.foo.logs               = + log[Console]
```

A general rule of thumb: With the presence of a new *log(s)* configuration, without the '+', like the sample code in the previous 2 sections, the already assigned *logs* of the *log controller* will be discarded, while with the '+', the new *logs* will be added to the **existing** list of *logs*.

4.6.2.3.1 Configuration unchanged

According to the rule description, that means on the other hand, when the configuration remains unchanged (which is typical once a stable configuration has been defined), the use of '+' is not significant at all. No further action will be done and the already attached objects will still be attached. This avoids the loss of state of the attached objects.

4.6.2.3.2 Configuration changed

The typical configuration updates include changing the log destination, adding log destination and the modifying properties of an existing log.
The first 2 cases are straightforward, as described at the end of Section 4.4.6. Imagine you have already assigned *logX* to a *location*. To **change** the destination to *logY* in the next reloading, do the following in the configuration file:

```
com.sap.foo.logs              = logY
```

logX is removed.
To **add to** the existing *logX*:

```
com.sap.foo.logs              = + logY
```

Both *logX* and *logY* persist.

The last type of modification (**changing properties**) mainly applies to *FileLog* as it has a number of attributes. A *FileLog* is identified by its 'pattern', therefore, a new pattern simply means a brand new *log* and this behaves like the first 2 cases. The '+' is significant.
If the changes are on the other attributes, like 'limit', 'cnt'. The '+' is not significant, no new *FileLog* will be created. The existing *FileLog* remains and will be modified with the corresponding new attribute values. Recommendations for this siutation is not to use the '+', or use it for *all* occurrences of the log assignments.

4.6.2.4 Summary highlights

The syntax rule described in the Javadoc API helps you to validate your syntax, but perhaps it is not too clear what attributes/values can be paired/assigned with/to what objects. Or what are required, what are optional (refer to Appendix C for default values). The summary below highlights the semantics. Note that the list is not exhaustive, a good trick is referring to the Javadoc API for a reasonable semantics.

Class	***Possible Attribute***	***Reference to API***	***Multiple entries***	***Required***
LogController		• *General for both Location, Category*		
	severity	<logController>.setEffectiveSeverity(int)		
	effSeverity	(same as 'severity')		
	minSeverity	<logController>.setMinimumSeverity(int)		
	maxSeverity	<logController>.setMaximumSeverity(int)		
	logs	<logController>.addLog(Log)	X	
	localLogs	<logController>.addLocalLog(Log)	X	
	privateLogs	<logController>.addPrivateLog(Log)	X	
	filters	<logController>.addFilter(Filter)	X	
	bundleName	<logController>.setResourceBundleName(String)		
Log		• *General for all Log subclasses*		
	severity	<log>.setEffectiveSeverity(int)		
	effSeverity	(same as 'severity')		
	encoding	<log>.setEncoding(String)		
	formatter	<log>.setFormatter(Formatter)		
	filters	<log>.addFilter(Filter)	X	
FileLog	pattern	new FileLog(String) //Filename		X
	limit	new FileLog(String, int, int)		
	cnt	(same as 'limit')		
Formatter	pattern	<formatter>.setPattern(String) //message format		

Table 4-5 Summary of configurable attributes of some classes

For primitive data type, you can easily assign an integer or string as an attribute value (except Severity, see below). For attribute that takes a class, as listed above, such as Log, Formatter, you will either use the classname directly (using default setting, see Section 4.6.2.1) or use the notation of a 'variable' (see Section 4.6.2.2).
For the case of interface *Filter*, assign the fully qualified classname of the user-defined class (make sure the class is already included in the classpath, and use an appropriate class loader). As a matter of fact, this applies to user-defined classes for *Log* and *Formatter* as well: use a full classname in case you are not using the standard classes provided by the SAP API.

Attribute	*Type*	*Possible Value*	*Description*
severity, effSeverity, minSeverity, maxSeverity	Integer	NONE	But constants defined in class *Severity* for your convenience.
		FATAL	Refer to Javadoc of class *Severity.*
		ERROR	All are case-sensitive.
		WARNING	
		INFO	
		PATH	
		DEBUG	
		ALL	
logs (classname)	Log	FileLog	Again, case-sensitive
		ConsoleLog	
logs (variable)		log[<id>]	<id> can be any logical, meaningful name
formatter (classname)	Formatter	TraceFormatter	
		ListFormatter	
		XMLFormatter	
formatter (variable)		formatter[<id>]	<id> can be any logical, meaningful name
Filters (classname)	*Filter* (interface)	com.sap.um.logging.Filter	Actual implementation defined by users. The arbitrary value shown on the left column indicates a class named 'Filter' is defined by user under the package 'com.sap.um.logging'.

If you understand the concept of relative severities (refer to Section 4.5.3) and would like to use it, this can be expressed in the properties file as well. Again, use the notation of square brackets.

Class	*Attricute of Relative severity*	*Reference to API*
LogController		• *General for both Location, Category*
	severity[<full name of relative l*ocation* or *category*>]	<logController>.setEffectiveSeverity(Location, int) <logController>.setEffectiveSeverity(Category, int)
	effSeverity[<full name of relative l*ocation* or *category*>]	(same as 'severity')
	minSeverity[<full name of relative l*ocation* or *category*>]	<logController>.setMinimumSeverity(Location, int) <logController>.setMinimumSeverity(Category, int)
	maxSeverity[<full name of relative l*ocation* or *category*>]	<logController>.setMaximumSeverity(Location, int) <logController>.setMaximumSeverity(Category, int)

Eg. *Category* "/System/Infrastructure/DB" establishes a relative severity with *location* "`com.sap.fooPackage.ClassX`".
With coding API:

```
Location _loc =
    Location.getLocation("com.sap.fooPackage.ClassX");
Category  cat =
    Category.getCategory("/System/Infrastructure/Database");
_cat.setEffectiveSeverity(_loc, Severity.WARNING);
```

Equivalence in properties file:

```
System/Infrastructure/DB.severity[com.sap.fooPackage.ClassX]
                                                 = WARNING
```

4.6.2.5 Notes

- What if the content of the properties file is invalid?
  Normally, if the syntax or semantics is incorrect, or there is simply a typo, the configuration will fail on that definition and prompt you an error message, explaining what is going wrong and may take the default setting (see **Appendix C**). The tool will continue to evaluate the subsequent configuration and output messages accordingly, based on whatever successful configuration that has been made.
  (The behavior results from the error handling mechanism designed in this tool. Refer to Section 5.1)
- Automatic lookup of default properties file?
  No. There is no system property to define this, nor there is a default physical file location specified for this. It is necessary to make one call in the program to load the properties. This could be improved when a *log console* (section 6.2) is ready.
- Can we define *Location* and/or *Category* object in the file?
  First of all, users should not be stuck with the concept of *creating* a category or location object and should not feel obliged to *create* one and *maintain* it. These are always accessible by making calls like `Category.getCategory()` and `Location.getLocation()` and the rest will be taken care of by the logging framework. It is simply a good practice to use a static variable to store the handle for better performance in case the source objects have to be accessed frequently. Thus, it is not necessary to explicitly define/create these source objects in the configuration file in advance; just use it directly in your program.

  However, just a side note of interest, when you are specifying certain properties for a source object in the configuration file, e.g. assigning severity or logs to a *location* or *category*, the corresponding *location* or *category* will be implicitly instantiated and correctly configured.
- Does the order of configuration matter?
  In general: NO.
  Not for class variable definition. Consider the following, it is valid and correct.

```
com.sap.foo.logs                = + log[Console], log[File]
log[Console]                          = ConsoleLog
log[Console].formatter                = ListFormatter
log[File]                             = FileLog
log[File].pattern                     = %t/trace.log
log[File].formatter                   = formatter[Trace]
formatter[Trace]                      = TraceFormatter
formatter[Trace].pattern              = %s: %24d: %m
```

Interestingly, order of severity assignment is not affected either. The severity inheritance is controlled by the closest ancestor who has severity explicitly assigned.

```
com.sap.foo.fooChild.severity    = INFO
com.sap.foo.severity       = WARNING
```

Imagine there are further descendants of `com.sap.foo.fooChild`, whom do not have explicit severity assignment. With this sequence of configuration, what are their severities? In this case, they actually inherit the severity from `com.sap.foo.fooChild`, not `com.sap.foo`, so the answer is INFO.

For other straightforward type of assignments, the order does matter. Apparently, the latter assignment will win. Easy example:

```
   log[Console]                        = ConsoleLog
   log[Console].formatter              = ListFormatter
   log[Console].formatter              = XMLFormatter
```

What will be the format of the output shown in the Console?

Refer to Section 4.6.2.3, the use of a '+' sign does not apply on log assignments specified in the current file configuration content (it is affecting the already **existing** assigned logs, such as from previous round of configuration in the case of periodic configuration). So, the sequence still matters, and *FileLog* will win (assuming no previous configuration has been done).

```
   com.sap.foo.logs                = + ConsoleLog
   com.sap.foo.logs                = + FileLog
```

If you want to assign multiple logs, simply use a comma as a delimiter:

```
 com.sap.foo.logs                  = + ConsoleLog, FileLog
```

5. Administrative issue

5.1 Error handling mechanism

Reason

The error handling mechanism should be done carefully in the logging tool. In any case, it should not jeopardize the execution of the running application. Therefore, any error occurred within the logging framework should be thrown and caught appropriately, be it a careless configuration error or an unexpected system error.

Workaround

The approach to deal with this is to tolerate exception and to correct the error as best as we can guess. The program will then continue to run. The exception is not lost; it is stored temporarily and can be retrieved by the caller through the methods `getException()` and `throwException()`. In addition, we also make use of our own logging framework to log the exception message to the internal dedicated *category* 'System/Logging', normally, with severity *error* and by default, output to the Console (or you should know by now how to redirect the messages to a destination other than the Console). Thus, users can get a better picture about the reason for certain not so standard behavior while the application is still running smoothly. This proves to be very robust for runtime exceptions such as `java.lang.IllegalArgumentException`, `java.lang.NullPointerException`, ….

Examples

A few potential mistakes are invalid severity definition, poor configuration, meaningless null message string, wrong resource bundle information, etc. These can be rectified internally and a message is logged to indicate the problem without terminating the program. There are several ways to replace the erroneous value with the best-guess value: using default value (refer to **Appendix C**) or estimating the closest value or simply ignoring it. For example:

- Invalid configuration for *FileLog* for the pair of parameters *limit* and *cnt*. The default setting for *FileLog* will be used, that is, *limit* and *cnt* will be both equal to zero.
- For some reasons, if severity level has been out of the range of the defined *Severity.MAX* and *Severity.MIN*, severity will be corrected to the closest limit *(MIN* or *MAX)*
- If a typo occurs in defining a *Log* or *Formatter*, say in the configuration file, the value will be ignored and nothing will be assigned
- A message string or message code should not be a null string, but when this happens, a constant '<null>' will be assigned.
- When a message code cannot be resolved to a localized string, the value of message code will be used directly

Note: exceptions generated from methods that are called in static field declarations and initializers do lead to immediate program termination upon class loading. Therefore, errors like this can be cited before software distribution.

5.2 Switching on/off the tool

Due to various reasons (eg. performance), logging is not turned on until requested by the administrator at the time of application deployment. By and large, this logging tool should not hinder your application performance of your application, but by default, logging is 'switched off' (This is achieved by the default severity level *Severity.NONE* assigned to all log controllers). This aligns with the purpose of decoupling the process of inserting logging code and the task of producing the actual logs/traces.

Eventually, this can be controlled through the *log console*. Currently, with the lack of interactive user interface, this can still be achieved by the feature of reloading the configuration file periodically. Initially, severity level is *NONE* (and this can be set 'globally' in the configuration file easily, making use of the hierarchical structure of the *log controllers*). With the specified periodic value to reload the configuration file regularly (see Section 4.6.1.2), users can adjust the severity level whenever they want to switch on the logging functions. This will automatically picked up a the next reloading and configuration will become effective to start producing the desired type and amount of messages.
(By the same token, you can always switch off the tool using the same technique.)

5.3 Compatibility with SUN JSR47

With the logging specification from SUN Microsystems, JSR47, there exists an implementation *'java.util.logging'* package starting from JDK1.4, or some other third-party software which is compliant to the specification. Although SAP logging API should be the standard tool to be used by the SAP projects, there is a demand for a wrapper to the standard API.
We understand the standard API can be a prerequisite for certain groups because third-party components are involved or that the SAP components to be run in a non-SAP environments, therefore we are providing a wrapper so that users can continue to use this standard API in their coding. It also provides the flexibility to stick with the standard JDK or to take advantage of the SAP logging tool features at a later time.

For users who are interested in a compatible API and are still working in JDK1.3, we have a wrapper to assist a smooth migration from 1.3 to 1.4. This wrapper is partially mapping the SUN standard, mainly focusing on leaving your development code intact, with minimal penalty on code modification. Howevre, currently, the *'loggingWrapper.jar'* is not publicly available.

Another wrapper implementation has been done for JDK1.4 developers to internally map to the SAP logging API and objects. It is more versatile to use this wrapper for the standard API as you have a choice to mingle the use of both the standard API and SAP API or simply stick with one API. Coding and configuration can be reused readily with minimal penalty.
For more information about this, please refer to the Javadoc for this package: *com.sap.tc.loggingWrapper*. Currently, the *'loggingWrapper.jar'* is not publicly available.

6. Future Development

6.1 Standard definition and rules

This will be an SAP-wide issue.
This involves, for example, predefined SAP *Categories*, location of output files, configuration files…

A rich set of standard *Category* should be available so that users can use readily. This includes the typical logical areas like database, network, security, etc. These predefined *categories* will become 'globally' available and the standard for various groups at SAP. This will be very beneficial because users do not have to randomly come up with a name, and need to worry about invalid configuration. The maintenance will become more organized.
A good set of categories needs to be defined carefully to cover the common requirements of most of the groups. Feedback and contribution from service group will be greatly helpful.

Other rules, such as standard locations of trace/log files, configuration files ….etc should also be determined. This will be decided upon by an expert group being put together at the moment.

6.2 Log console

This provides an interactive interface for users to easily manipulate the logging tool to suit their needs, on both the logging behavior and the logging output. The current list of tasks is:

- A message viewing tool
- Options to allow selectively display trace and log messages
- Interactive configuration (including starting and stopping the tool)
- Support message translation at the time of viewing

This will be started by the InQMy group for the InQMy server.

6.3 Thread-specific tracing

Thread-specific and business-process-specific tracing is a well-known problem with server-based systems. The execution of a single request might span across several servers (threads) and cause problems. It is tricky to trace and analyze such request.

Even if it is possible to tightly time a temporary modification of the global settings (within one VM) of the Logging API, you usually get lots of messages from other requests that you are not interested in. This hurts the system performance as well.

Thread-specific tracing allows the configuration specific to a single thread: above all setting a lower severity for tracing in that single thread. In principle, this is done by putting these parts of the configuration into a *ThreadLocal*. Based on these thread-specific settings, settings specific to a request spanning several servers (threads) and even systems can be implemented (business-process-specific tracing).

To this end, the first thread in the chain packs its settings (which it gets from the front-end or a corresponding properties file) into a *transfer object*. It then hands over this transfer object as an argument to subsequent threads executing sub-requests. These threads in turn register the transfer object with the Logging API, thus inheriting settings from callers. After finishing the request, the Logging API is called again to forget about the settings specified via the transfer objects.

Appendix A

Severity Levels

The following are the different levels of severity, in ascending order:

Severity	Value	Details
MIN	0	Minimum restriction
DEBUG	100	For debugging purpose, with extensive and low level information
PATH	200	For tracing the execution flow, e.g. used in the context of entering and leaving a method, looping and branching operations
INFO	300	Informational text, mostly for echoing what has been performed
WARNING	400	Application can recover from anomaly, and fulfill the desired task, but need attention from developer/operator
ERROR	500	Application can recover from error, but cannot fulfill desired task due to the error
FATAL	600	Application cannot recover from error, and the severe situation causes fatal termination,
MAX	700	Maximum restriction

Table A-1 Definition of severity levels

Here are the other 2 useful severity keywords that will totally enable and disable tracing/logging:

Severity	Value	Details
ALL	0	Output messages of all severity
NONE	701	No messages will be logged

Appendix B

Overview of Classes Hierarchy

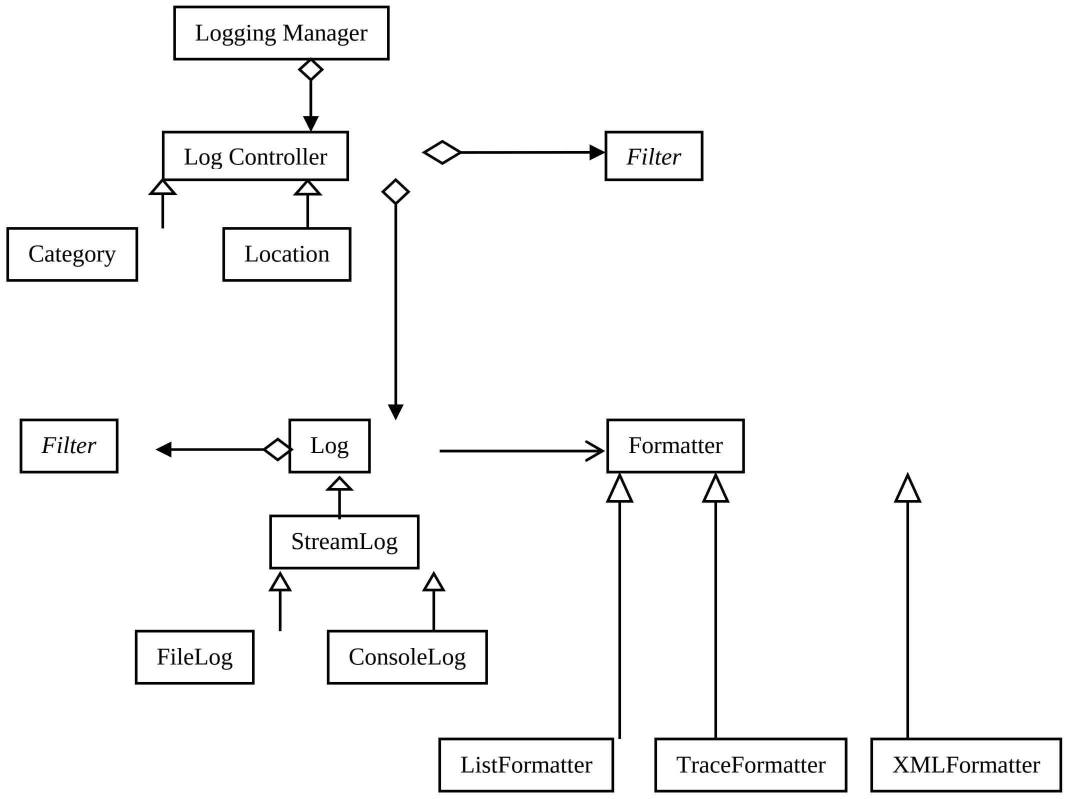


Figure B-1 Class diagram of the logging API

Appendix C

Default behavior of certain objects

Class	***Attribute***	***Default Value***	***Description***
LogController	Severity	Severity.NONE	No output at all
	MinSeverity	Severity.ALL	No floor restriction: out all
	MaxSeverity	Severity.NONE	No ceiling restriction: output nothing
	<output logs>	<none>	No output at all
	<filters>	<none>	No further filtering
Log	Severity	Severity.ALL	Do not further suppress messages
	<filters>	<none>	No further filtering
ConsoleLog	Formatter	TraceFormatter	
FileLog	Formatter	ListFormatter	
	Limit	0	No file size limit. 'Count' must == 0. No rotating set of numbered files.
	Count	0	Logfile name generated without a sequence number.
TraceFormatter	Pattern	%24d %-40l [%t] %s: %m	Jan 01, 2001 10:10:00 PM com.sap.FooClass.fooMethod [main] Fatal: A sample fatal message
ListFormatter	Pattern		#1.3#10.48.27.165:4A5AB2:E99D42D4F4:-8000#Mon Jan 01 22:00:00 PDT 2001#com.sap.FooClass#com.sap.FooClass.fooMethod#main##0#0#Fatal##Plain###A sample fatal message#
XMLFormatter	Pattern	<!ELEMENT log (record*)> <!ELEMENT table (id, time, source, severity, location, thread, msg-type, msg-code?, bundle?, msg-clear?, args?)> <!ELEMENT id (#PCDATA)> <!ELEMENT time (#PCDATA)> <!ELEMENT source (#PCDATA)> <!ELEMENT severity (#PCDATA)> <!ELEMENT location (#PCDATA)> <!ELEMENT thread (#PCDATA)> <!ELEMENT msg-type (#PCDATA)> <!ELEMENT msg-code (#PCDATA)> <!ELEMENT bundle (#PCDATA)> <!ELEMENT msg-clear (#PCDATA)> <!ELEMENT args (arg+)> <!ELEMENT arg (#PCDATA)>	<record> <id>10.48.27.165:4A5AB2:E99D2EDAFF:-8000</id> <time>Mon Jan 01 22:00:00 PDT 2001</time> <source>com.sap.FooClass</source> <location>com.sap.FooClass.fooMethod</location> <thread>main</thread> <severity>Fatal</severity> <msg-type>Plain</msg-type> <msg-clear>A sample fatal message</msg-clear> </record>

Table C-1 Default attributes values

Class	***Property***	***Default Value***	***Description***
Location	*root name*	“”	Empty string. Full name should be hierarchical, delimited by “.”
Category	*root name*	“/”	Similar to root directory. Full name should be hierarchical, delimited by “/”
FileLog	*Filename, with size limit and count*	<filename>.<ext>.cnt	Eg. trace.log.0, trace.log.1, …
Trace	*Filename, if not specified by user*	%t/trace.log	Eg. C:\temp\trace.log
TraceFormatter	*Message displayed*	Content of ‘MsgClear’ in any case	Either translated text in ‘MsgClear’ or text provided by users explicitly in ‘MsgClear’
ListFormatter	*Message displayed*	Optional	Both MsgCode & MsgClear available, up to the viewer
XMLFormatter	*Message displayed*	Optional	Both MsgCode & MsgClear available, up to the viewer
Location or Category with relative ‘A’	*Output Behavior by relative*	Nothing will be output by relative ‘A’	Relative ‘A’ unaffected, and still has default Severity.NONE and no output destination. Only when explicit configuration is done on relative ‘A’, then output will be generated from it

Table C-2 Highlights of default behavior