Contents

1 Java Logging API

1.1 Introduction

1. **logging**: the process of writing log messages during the execution of a program to a central place.
2. **utility**: report and persist error, warning and info messages (*e.g. runtime statistics*) to retrieve them later for analysis.

1.2 Logging in Java

1. **package**: `java.util.logging`.
2. **key concepts**:
   - `Logger`: a class that models logger objects used to log messages for a specific system or application component.
   - `LogRecord`: a class that models log records used to pass requests between the logging framework and individual log handlers.
   - `Handler`: a class that models a handler used to export log records to a variety of destinations (*memory, output streams, consoles, files, and sockets*).
   - `Level`: a class that defines a set of standard logging levels that can be used to control logging output for a specific program.
   - `Filter`: an interface that defines a fine-grained control over what gets logged, beyond the control provided by log levels.
   - `Formatter`: a class that models objects to format the log records.

1.3 Logger naming and creation

1. **naming**:
    - loggers are normally named, using a hierarchical dot-separated namespace.
    - names can be arbitrary strings, but they should be based on the package/class name of the logged component.
    - it is possible to create anonymous loggers that not stored in the Logger namespace.
2. **creation**:
    - using factory methods to either create a new logger or return a suitable existing Logger.
    - a logger returned by a factory method may be garbage collected if it's not strongly referenced.
3. **logger hierarchy**:
    - **principle**: each logger keeps track of its “parent” Logger, which is its nearest existing ancestor in the Logger namespace.
    - **example**: a logger from the `com.example` key is a child of `com` Logger, which in turn is a child of the empty String.

```
import java.util.logging.Logger;

// assume the current class for which to created a logger is called MyLogger
private static final Logger LOGGER = Logger.getLogger(MyLogger.class.getName());
```

1.4 Log records

1. **definition**:
    - a `LogRecord` object is used to pass logging requests between the logging framework and individual log handlers.
    - once passed to the framework, the record should no longer be updated by the client application.
2. **notes**:
    - if the client application used a log method that doesn't explicitly provide a source class name or method, the log record class will infer them automatically when they are first accessed (by invoking `getSourceClassName()` or `getSourceMethodName()`) by analyzing the call stack.
    - a log record is serializable
    - parameters of the log message are written as their corresponding Strings during serialization
    - resource bundles instances are not serialized, only their names.

1.5 Levels

1. **definition**:
   - each logger has a `Level` associated to it, reflecting the minimum level of messages that the logger cares about, called "**message severity**".
   - enabling logging at a given level also enables logging at all higher levels.
2. **message severity levels**: ordered and specified by ordered integers:

Level	Description	Note
`Level.SEVERE`	log serious failure messages	*highest value*
`Level.WARNING`	log warning messages indicating a potential problem	
`Level.INFO`	log informational messages	
`Level.CONFIG`	log static configuration messages.	
`Level.FINE`	log tracing messages	
`Level.FINER`	log fairly detailed tracing messages	
`Level.FINEST`	log highly detailed tracing messages	*lowest value*
`Level.OFF`	turn logging off	
`Level.ALL`	log all levels of messages	

3. **levels and parents**: a logger having a Level set to `null` will effectively inherit the level of its ancestor, which may in turn be recursively obtained from its parent, and so on up the tree.
4. **level configuration**:
   - **logging configuration file**: a logging configuration file contains properties including the level associated to a logger, as described in the description of the `LogManager` class.
   - **dynamic configuration**: `public void Logger.setLevel(Level newLevel);`
   - **note**: changing the level of a logger can potentially affect the level of its children loggers.
5. **process**:
   - on each logging call, the logger initially performs a cheap check of the request level against its effective log level: if the request level is lower than the effective level of the logger, the logging call returns immediately.
   - after passing the initial cheap test, the logger will create a `LogRecord` to describe the logging message.
   - it will call a `Filter` (*if present*) afterwards to do a more detailed check on whether the record should be published:
     1. if that passes, it will publish the `LogRecord` to its output handlers.
     2. by default, it will also publish the `LogRecord` recursively to its parents' handlers up the tree.

```java
// SEVERE, WARNING and INFO messages will be logged
LOGGER.setLevel(Level.INFO);
```

1.6 Handlers

1. **definition**:
    - each logger can have access to several handlers.
    - the `java.util.logging.Handler` object receives the log message from the logger and exports it to a certain target.
    - formatting (*including localization*) is the responsibility of the output `Handler`, which will typically invoke a `Formatter`.
2. **examples of standard handlers**:
    - `java.util.logging.ConsoleHandler`: write the log message to the console.
    - `java.util.logging.FileHandler`: write the log message to a file.
3. **turning off a handler**: `handler.setLevel(Level.OFF);`
4. **note**: log messages having `INFO` level or higher will be automatically written to the console.

1.7 Filters

1. **definition**: a functional interface defining a filter that can be used to provide fine grain control over what is logged, beyond the control provided by log levels.
2. **process**:
    - each logger and each handler can have a filter associated to it.
    - the logger or handler invokes the `isLoggable(LogRecord record)` on each log record to check if it should be published or not.
3. **methods**:

```java
/*
* checks if a given log record should be published or not
*/
boolean isLoggable(LogRecord record);
```

1.8 Formatters

1. **definition**: each handler's log records can be configured with a formatter.
2. **available predefined formatters**:
    - `java.util.logging.SimpleFormatter`: generates all log records as text.
    - `java.util.logging.XMLFormatter`: generates XML output for the log records.

3. **creating custom formatters**:
    - extend the `java.util.logging.Formatter` abstract class.
    - implement `public String format(LogRecord record)` invoked on every log record.
4. **note**: formatting need not occur synchronously: it may be delayed until a `LogRecord` is actually written to an external sink.
5. **example**: creating an HTML formatter:

```java
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.logging.Formatter;
import java.util.logging.Handler;
import java.util.logging.Level;
import java.util.logging.LogRecord;

public class HTMLFormatter extends Formatter {

    // inherited abstract method to be implemented to define formatting behavior
    @Override
    public String format(LogRecord record) {
        StringBuffer buf = new StringBuffer(1000);
        buf.append("<tr>/n");

        // color any levels >= WARNING in red
        if(record.getLevel().intValue() >= Level.WARNING.intValue()) {
            buf.append("\t<td style=\"color:red\">");
            buf.append("<b>");
            buf.append(record.getLevel());
            buf.append("</b>");
        }
        else {
            buf.append("\t<td>");
            buf.append(record.getLevel());
        }

        buf.append("</td>\n");
        buf.append("\t<td>");
        buf.append(calcDate(record.getMillis()));
        buf.append("</td>\n");

        buf.append("\t<td>");
        buf.append(formatMessage(record));
        buf.append("</td>\n");

        buf.append("</tr>\n");
        return buf.toString();
    }

    private String calcDate(long millis) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("MMM dd,yyyy HH:mm");
        Date resultDate = new Date(millis);
        return dateFormat.format(resultDate);
    }

    // method invoked just after the formatter is created
    @Override
    public String getHead(Handler h) {
        StringBuffer buf = new StringBuffer(10000);
        buf.append("<!DOCTYPE html>\n");

        buf.append("\t<head>\n");
        buf.append("\t\t<style>\n");
        buf.append("\t\ttable { width: 100% }\n");
        buf.append("\t\tth { font: bold 10pt Tahoma; }\n");
        buf.append("\t\ttd { font: normal 10pt Tahoma; }\n");
        buf.append("\t\th1 { font: normal 11pt Tahoma; }\n");
        buf.append("\t\t</style>\n");
        buf.append("\t</head>\n");

        buf.append("\t<body>\n");
        buf.append("\t\t<h1>" + (new Date()) + "\n");
        buf.append("\t\t<table border=\"0\" cellpadding=\"5\" cellspacing=\"3\">\n");
        buf.append("\t\t\t<tr align=\"left\">\n");
        buf.append("\t\t\t\t<th style=\"width:10%\">LogLevel</th>\n");
        buf.append("\t\t\t\t<th style=\"width:15%\">Time</th>\n");
        buf.append("\t\t\t\t<th style=\"width:75%\">LogMessage</th>\n");
        buf.append("\t\t\t</tr>\n");

        return buf.toString();
    }

    // method invoked just after the formatter is closed
    @Override
    public String getTail(Handler h) {
        StringBuffer buf = new StringBuffer(100);
        buf.append("\t\t</table>\n");
        buf.append("\t</body>\n");
        buf.append("</html>");

        return buf.toString();
    }
```

```
}
```

1.9 Resource bundles

1. **principle**: each logger may have a `ResourceBundle` associated to it, used for localizing logging records.
2. **association**:
   - **by name**: `public static Logger getLogger(String name, String resourceBundleName);`
   - **by value**: `public void setResourceBundle(ResourceBundle bundle);`
3. **searching for a resource bundle**:
   - the logger will first look at whether a bundle was specified by value, if not it will check if it was specified by name.
   - if no bundle is specified by value or by name, then it will inherit that of its parent, recursively up the tree.
   - if the bundle is found by value (whether owned or inherited), then it will be used
   - if the bundle is found by name (whether owned or inherited), then it will be mapped to a `ResourceBundle` object using the default Locale at the time of logging.
4. **note**: if a logger doesn't have an associated `ResourceBundle`,

1.10 Logger methods

1. **methods that take a message argument**:
   - **message format**: raw value or a localization key
   - **process**: during formatting, if the logger has (*or inherits*) a `ResourceBundle`, and if that bundle has a mapping for the message `String`, then it'll be replaced by its localized value, otherwise the original `String` will be used.
2. **methods that take a message supplier argument**: the `Supplier<String>` function is used to construct the desired log message only when the message actually is to be logged based on the effective log level, thus eliminating unnecessary message construction.
3. **categories**:
   - `log` **methods**: take a log level, a message string, and optionally some parameters to the message string.
   - `logp` **methods** (*log precise*): like the `log` methods, but also take an explicit source class name and method name.
   - `logrb` **methods** (*log with resource bundle*): like the `logp` methods, but also take an explicit `ResourceBundle` object to localize the log message.
   - **convenience tracing methods**:

    1. the “entering” methods: tracing method entries.
    2. the “exiting” methods: tracing method returns.
    3. the “throwing” methods: tracing throwing exceptions.
  - **convenience methods for logging a string at a given level**:
    1. `public void severe(String msg)` or `public void severe(Supplier<String> msgSupplier)`.
    2. `public void warning(String msg)` or `public void warning(Supplier<String> msgSupplier)`.
    3. ...
4. **notes**:
   - methods that don’t take an explicit source class name and method name are “best effort” methods that rely on the framework automatically inferring information about the class and method called into the logging method, and remain approximate and potentially wrong.
   - all logger methods are multi-thread safe.

1.11 Log Manager

1. `LogManager` **class**:
   - create and manage the logger and the maintenance of its configuration.
   - manages a hierarchical namespace of `Logger` objects, such that all named loggers are stored within this namespace.
   - manages a set of logging control properties: a set of key-value pairs which can be used by `Handlers` and other logging objects to configure themselves.
   - a default configuration is provided.
2. **note**: all log manager methods are multi-thread safe.
3. **example**:

```
/*
* retrieving the global log manager object
* note: created during class initialization and cannot be changed subsequently
*/
LogManager globalLogManager = LogManager.getLogManager();

// set the logging level of all loggers to Level.FINE
LogManager.getLogManager()
.getLogger(Logger.GLOBAL_LOGGER_NAME)
.setLevel(Level.FINE);
```