**LOGGING:**

1. In programming, logging is the process of recording information about a program’s execution—like a digital diary for your code. It helps developers understand what’s happening inside an application, especially when things go wrong.

🛠️ Why logging matters

* Debugging: Track down bugs by seeing what the program was doing before it failed.
* Monitoring: Keep an eye on performance, usage patterns, or suspicious activity.
* Auditing: Record important events like user logins or data changes for compliance or security.

🧱 What gets logged?

* Errors and exceptions
* Warnings about potential issues
* Informational messages (e.g., “User X logged in”)
* Debug details (e.g., variable values, method calls)
* Logging in programming, refers to recording activities/events. Usually, the application developers should take care of logging.
* To make the job of logging easier, Java provides various frameworks − log4J, java.util.logging (JUL), tiny log, logback, etc

**SLF4J:**

SLF4J, which stands for Simple Logging Facade for Java, is not a logging framework itself. Instead, it's a facade (an abstraction layer) that provides a simple and unified API for various logging frameworks in Java.

**The Problem SLF4J Solves (The "Logging Hell" in Java):**

In the Java ecosystem, there are several popular logging frameworks like Log4j, Logback, and java.util.logging (Java's built-in logging). If your application directly used one of these, and then you decided to switch to another (e.g., for performance, features, or simply because a third-party library used a different one), you'd have to rewrite all your logging calls. This could be a huge pain, especially in large projects or when integrating multiple libraries**.**

**How SLF4J Works (The Facade Pattern):**

SLF4J acts as a "middleman" or "proxy." You write your application's logging code using the SLF4J API (e.g., org.slf4j.Logger, org.slf4j.LoggerFactory). At runtime, you then "bind" SLF4J to a specific underlying logging framework by including the appropriate SLF4J binding (a JAR file) in your project's classpath.

K**ey Benefits of Using SLF4J:**

1. **Decoupling (reducing/removing dependencies of various parts of program on another) :** Your application code becomes decoupled from the concrete logging implementation. You can switch between logging frameworks (e.g., from Log4j to Logback) at deployment time simply by changing the binding JARs, without modifying your application's source code.
2. **Flexibility:** It gives you the flexibility to choose the best logging framework for your specific needs without being locked into one.
3. **Consistency:** Libraries and frameworks can use SLF4J for their internal logging, ensuring that even if you use multiple libraries, their logs can be routed through your chosen logging framework and configuration.
4. **Performance:** SLF4J supports parameterized logging (e.g., logger.debug("Value of x is {}", x);). This is more efficient than concatenating strings (logger.debug("Value of x is " + x);) because the string concatenation only happens if the log level is enabled, avoiding unnecessary object creation and string operations when logging is disabled.
5. **Simplicity:** The SLF4J API is generally considered clean and easy to use

**Logging Framework Overview**

A logging framework usually contains three elements –

1. **Logger:** Captures the message along with the metadata.
2. **Formatter:** Formats the messages captured by the logger.
3. **Handler:** The Handler or appender finally dispatches the messages either by printing on the console or, by storing in the database or, by sending through an email.

Some frameworks combine the logger and appender elements to speed up the operations

**Logger Object**

To log a message, the application sends a logger object (sometimes along with the exceptions if any) with name and security level.

**Severity Level**

The messages logged will be of various levels. The following table lists down the general levels of logging.

Fatal: Severe issue that causes the application to terminate

ERROR: Runtime errors.

WARNING: In most cases, the errors are due to the usage of deprecated APIs.

INFO: Events that occur at runtime.

DEBUG: Information about the flow of the system.

TRACE: More detailed information about the flow of the system.

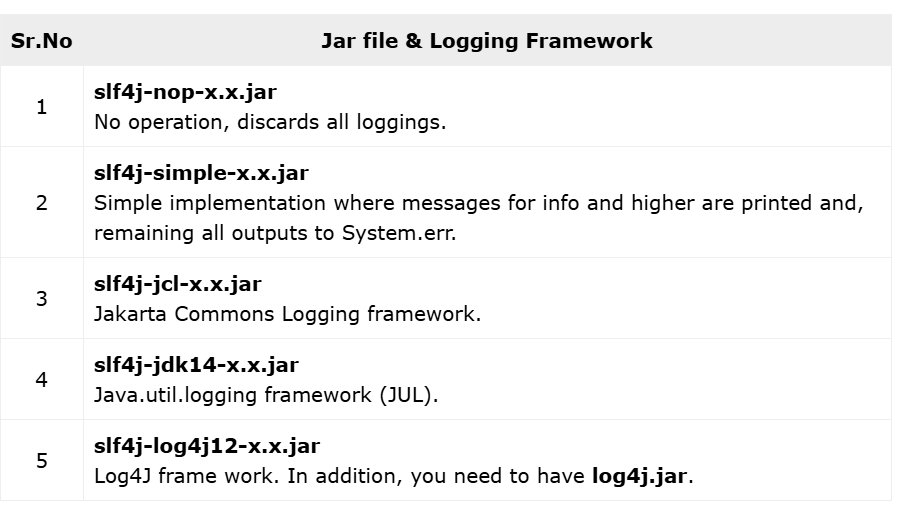
**ENVIRONMENT SETUP:**

There are 2 ways to set slf4j:

* 1. Working with Java Project
  2. Working with Maven Project

Java Project:

1. Download slf4j jar file
2. Set build path by creating a project (Add as external JARs in library tab)
3. In addition to slf4j-api.x.x.jar file, SLF4J provides several other Jar files as shown below. These are called SLF4J bindings.
4. Where each binding is for its respective logging framework.
5. The following table lists the SLF4J bindings and their corresponding frameworks.



To make SLF4J work along with slf4l-api-x.x.jar, you need to add the respective Jar file (binding) of the desired logger framework in the classpath of the project (set build path).

To switch from one framework to other, you need to replace the respective binding. If no bounding is found, it defaults to no-operation mode.

**Maven Project: (Recommended and Mostly used)**

1. Create a maven project
2. Edit pom.xml by adding slf4j-api dependencies and LoggingBack binding dependency( mostly used and recommended)

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>my-slf4j-app</artifactId>

<version>0.0.1-SNAPSHOT</version>

<properties>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

<slf4j.version>1.7.36</slf4j.version> <logback.version>1.2.11</logback.version> </properties>

<dependencies>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>${slf4j.version}</version>

</dependency>

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>${logback.version}</version>

</dependency>

</dependencies>

</project>

1. **Update Maven Project:**

* Save the pom.xml file.
* In the Eclipse Package Explorer, right-click on your project.
* Select Maven > Update Project...
* Ensure your project is selected and click OK. Maven will download the required JARs and add them to your project's class path automatically.(hence, not reqd to install manually)
* These downloaded JARs are stored in your **local Maven repository**, which is typically located in your user home directory under .m2/repository.
* Maven then **adds these JARs to your project's classpath** for compilation, testing, and execution.

Now Steps are same for both

1. Write Code
2. Configure Your Logging Framework (if necessary):
   1. If you're using Logback, Log4j 2, or Log4j 1.x, you'll typically need a configuration file (e.g., logback.xml, log4j2.xml, or log4j.properties) placed in your project's src folder (or a folder added to the classpath) to define log levels, appenders (where logs go, e.g., console, file), and formats.
3. Generally placed config file at location : src/main/resources.

Example**: LogBack.xml config file**

<configuration>

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>

</encoder>

</appender>

<root level="debug">

<appender-ref ref="STDOUT" />

</root>

</configuration>

**CONTENTS OF POM.XML:**

Use of <properties> for Versions:

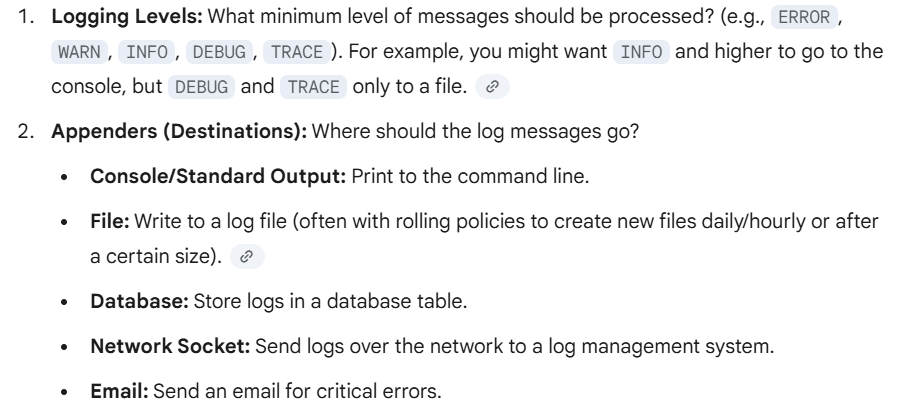
* Your pom.xml: Hardcodes the version number directly in the <version> tag of the dependency.
* My pom.xml: Defines versions in a <properties> section (e.g., <slf4j.version>1.7.36</slf4j.version>) and then references them using ${propertyName} (e.g., ${slf4j.version}).

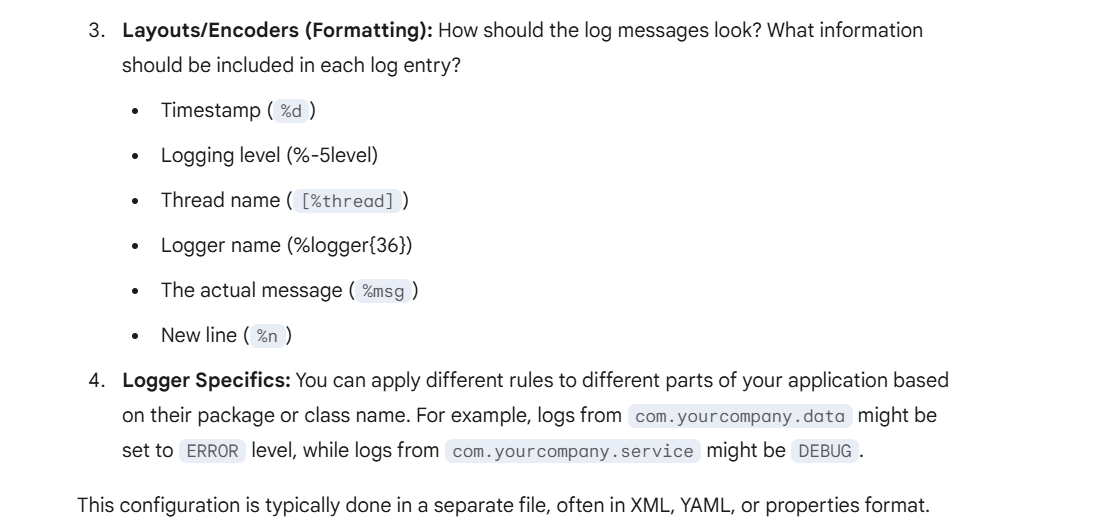
Why this is good practice:

* **Centralized Management:** If you have many dependencies that share the same version (e.g., multiple SLF4J or Spring modules), you only need to change the version in one place (the <properties> section) instead of searching and replacing it in multiple dependency declarations.
* **Readability:** It can make the dependencies section cleaner, especially for large projects.
* First dependency is for slf4j
* Second is for logback logging

**Meaning of configuring:**

Configuring your logging framework" means telling the *actual logging library* (like Logback, Log4j 2, or java.util.logging) how you want it to behave. It's about defining rules for:





**It's necessary whenever you want to customize how your logs are handled.** If you just want basic console output and the default level suits you, then technically you might skip it, but that's rare for anything beyond the simplest examples.

For Maven projects, you typically place these configuration files in the src/main/resources folder (or src/test/resources for test-specific configurations). When Maven builds your project, it automatically copies the contents of src/main/resources into the root of your compiled JAR/WAR file.

This means that when your application runs, the logging framework can find its configuration file on the classpath, and then it will apply the rules defined within it.

**Issues if logging is not added in pom.xml or as class path in java Projs:**

The binding resolution happens at runtime. SLF4J dynamically discovers which logging implementation (like Logback or Log4j 2) is available on the classpath.

However, for that runtime resolution to work, the binding JAR file (and the actual logging framework JARs) must *be present* on the classpath at runtime.

Here's why you need to include the binding in your pom.xml (or manually add the JARs) **:**

1. **For Maven to Package It:** When you build your project with Maven (mvn package), it will create a JAR or WAR file that bundles all your project's code and its dependencies. If the binding JAR is not listed in your pom.xml, Maven will not include it in the final artifact.
   * **Result:** When you try to run that packaged application, the SLF4J facade will execute, but it won't find any concrete logging implementation.
2. **To Avoid the NOP Logger:** If SLF4J's LoggerFactory cannot find a suitable binding it will default to a **No-Operation (NOP) logger**.
   * **What this means:** All your logger.info(), logger.debug(), logger.error() calls will effectively do nothing. No messages will be printed to the console, files, or any other configured output. Your application will run without errors (because SLF4J gracefully handles the missing binding), but you won't see any logs. You'll often see a warning message like:
   * SLF4J: No SLF4J providers were found.
   * SLF4J: Defaulting to no-operation (NOP) logger implementation
   * SLF4J: See http://www.slf4j.org/codes.html#noProviders for further details.
3. **For Development and Testing:** During development, you'll want to see your log messages immediately to debug and understand your application's behavior. Without the binding, you'd be developing "blind" from a logging perspective.

**Logger:**

A logger is an object used in programming to record messages about what your application is doing. Think of it as your app’s personal narrator—it keeps track of events, errors, and other useful information while the program runs.

🧰 What does a logger do?

* Captures messages at different severity levels (like INFO, DEBUG, ERROR)
* Sends logs to various outputs (console, files, remote servers, etc.)
* Formats messages for readability or structured analysis
* Filters logs based on level or source

🪵 Logging vs. Printing

While System.out.println() or print() might seem easier, logging frameworks offer:

* **Log levels** (e.g., DEBUG, INFO, WARN, ERROR)
* **Output control** (write to files, databases, or monitoring tools)
* **Formatting and filtering** for better readability and analysis

**Logger Interface:**

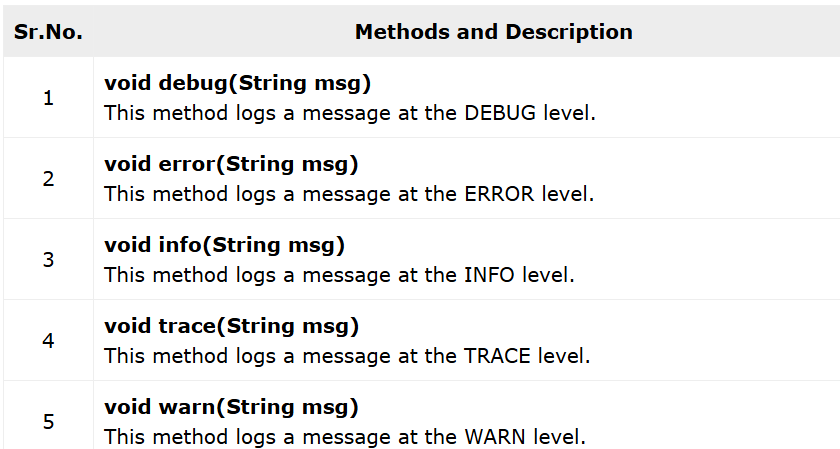
📋 1. **Logger Interface**

A **Logger interface** defines a common contract for logging behavior. It abstracts logging so your application isn’t tied to a specific logging implementation.

🧰 Purpose:

* Define methods like info(), debug(), warn(), and error() without dictating how they should work internally.
* Enable multiple logging libraries (like Log4j, SLF4J, etc.) to implement the same interface.

The following are methods in logger interface:



**LoggerFactory Class // creates object for logger interface**

The LoggerFactory class of the org.slf4j package is a utility class, which is used to generate loggers for various logging APIs such as log4j, JUL, NOP and simple logger.

**Logger getLogger(String name)**

This method accepts a string value representing a name and returns a **Logger** object with the specified name.

Logger logger = Logger.getLogger("MyAppLogger");

"MyAppLogger" is a name (just a string) that uniquely identifies your logger.

* The method looks up if a logger with that name already exists:
* If **yes**, it gives you the existing logger.
* If **no**, it **creates a new Logger object**, assigns it the name "MyAppLogger", and returns that.

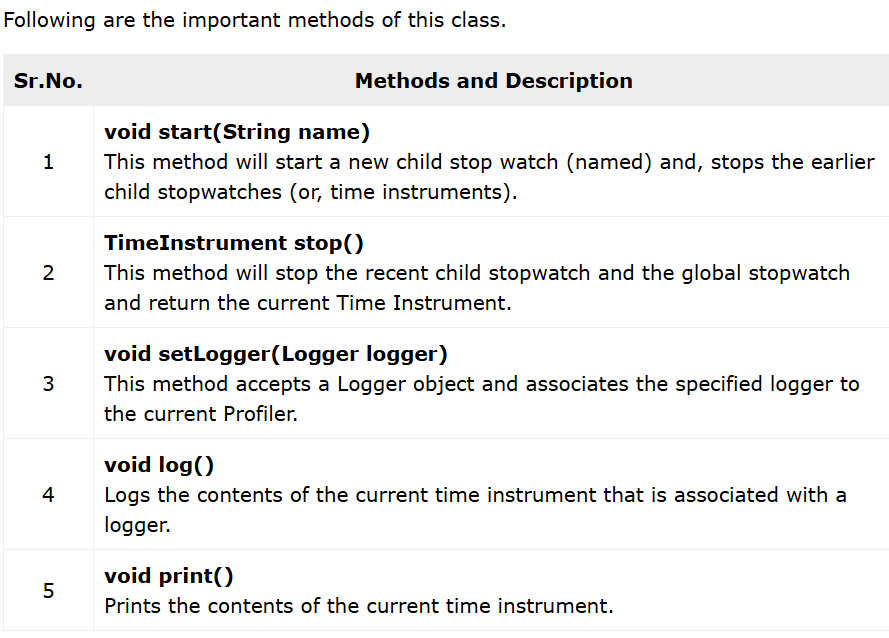
A **Logger Factory Class** (or simply "Logger Factory") is a design pattern (often a Factory or Abstract Factory) responsible for creating and providing instances of logger implementations. It acts as a centralized point for obtaining logger objects, ensuring consistency and allowing for configuration

🧰 Purpose:

* Centralizes logger creation logic.
* Caches or reuses existing loggers for the same name.

**Profiler Class**

This class belongs to the package org.slf4j. It is a performance analysis tool that helps measure how long different parts of your application take to run.



🧰 Purpose:

* Track execution time of methods, blocks of code, or API calls.
* Help optimize performance bottlenecks.