**Log4J and SLF4J Overview**

**✅ What is Logging?**

* **Definition:**  
  Logging is the process of keeping track of an application's flow of execution by recording log messages.
* **Purpose:**  
  These log messages help developers and support teams to:
  + Trace the state of the application at any point in time (including past).
  + Debug and identify issues in production or development.
  + Understand the flow of execution, including which components and modules are involved.
* **Example:**
  + Which class was called?
  + What method was executed?
  + What input/output data was processed?
  + What exceptions occurred?

**✅ What is Auditing?**

* **Definition:**  
  Auditing is the process of recording **user activities** performed within the application.
* **Purpose:**  
  It helps track "who did what and when" inside the system.
* **Examples of Auditable Events:**
  + User signed in
  + Opened inbox
  + Replied to a mail
  + Signed out

**✅ Difference between Logging and Auditing**

| **Aspect** | **Logging** | **Auditing** |
| --- | --- | --- |
| Focus | Tracks application flow and system state | Tracks user activities and actions |
| Purpose | Debugging, monitoring, tracing | Security, compliance, activity tracking |
| Example | Method calls, exceptions, system messages | Login, file access, dat |

## ✅ **Use Cases of Logging**

### 🔍 ****Why Logging is Important in a Project****

1. **During Unit Testing**
   * When test cases fail, developers use log messages to **debug** and trace the cause of the failure.
   * Helps quickly identify the method, input, or exception that caused the issue.
2. **While Fixing Bugs Reported by Testers**
   * Developers need to understand **how the application was executing** at the point the bug occurred.
   * Logs reveal the **flow of execution** and help reproduce the issue.
3. **After Project Release (Production Bugs)**
   * Bugs reported by end users (via onsite team or client org) are analyzed using **log files**.
   * Offshore teams depend on these logs to understand the **production environment behavior** when the issue happened.
4. **In Production Maintenance**
   * If the application **suddenly crashes** or behaves unexpectedly, logs are the primary source to identify the cause.
   * Logs are used to investigate **exceptions, memory leaks, or DB crashes**.
5. **Database Backup & Recovery**
   * While backing up or restoring the database software (DB s/w), logs are crucial to:
     + Trace errors
     + Track the process flow
     + Ensure data consistency
6. **Transaction Support**
   * During **financial or sensitive operations**, logs are required to:
     + Trace every step of the transaction
     + Rollback or recover if failure happens
     + Ensure audit compliance

## 🌐 **Project Teams: Onsite vs Offshore**

* **Onsite Team**  
  Located at the client organization; responsible for installing, configuring, and interacting directly with end users.
* **Offshore Team**  
  Based in the software company; develops the application and supports maintenance remotely by analyzing logs and fixing issues.

## 🌎 **Project Environments**

Every software project typically includes the following environments:

| **Environment** | **Ownership** | **Purpose** |
| --- | --- | --- |
| Dev Environment | Software Company | Application development and initial testing |
| Testing Environment | Software Company | Manual/automated testing, bug fixing |
| UAT Environment | Client Organization | Final testing by the client |
| Production | Client Organization | Live application used by end users |

**Note:**  
Logging code is **implemented during development**, but logs are **used across all environments** (especially UAT and production).

## ⚠️ **Limitations of Using** System.out.println() **or** System.err.println() **for Logging**

| **Limitation** | **Explanation** |
| --- | --- |
| ❌ Console only | Logs are shown on the monitor and lost after time |
| ❌ No categorization | Cannot separate info, warning, debug, error messages |
| ❌ No formatting | No structure or readability in logs |
| ❌ No multi-destination support | Cannot write logs to files, DB, mail servers, etc. |
| ❌ No archival or history | Cannot access logs from a specific date/time |
| ❌ No filtering | Cannot extract logs based on severity, component, or module |
| ❌ Not thread-safe | System.out.println() is single-threaded and unsuitable for multi-threaded (especially web) applications |

## ✅ **Recommended Logging Frameworks**

To overcome these limitations, Java projects typically use:

* **Log4j / Log4j2** – Full-featured, configurable logging framework.
* **SLF4J (Simple Logging Facade for Java)** – Acts as a facade or abstraction over various logging frameworks (Log4j, Logback, java.util.logging).
* **Logback** – Successor of Log4j, integrates well with SLF4J.

### 🔧 ****Problem with Traditional Debugging****

To overcome the limitations of System.out.println() for debugging and tracking, we use:

1. **Java Assertions** (from Sun JDK)
2. **Java Logging API** (java.util.logging)
3. **Commons Logging** (Apache)
4. **JBoss Logging** (Red Hat)
5. **Log4j** (Apache - ✅ Best in market)
6. **Logback** (Adobe)

### 📘 ****SLF4J**** (Simple Logging Facade for Java)

* Acts as a **unified abstraction layer** for various logging frameworks.
* Supports:
  + **Log4j**
  + **Logback**
  + **Commons Logging**
* Allows us to switch underlying logging implementation without changing logging code.

### 🪵 ****Log4j Overview****

* **Type**: Logging framework for Java
* **Vendor**: Apache
* **Open Source**: Yes
* **Stable Version**: 1.x (e.g., 1.2.17)
* **Unstable Version**: 2.x (in some environments)

🔗 **Download**:  
[Log4j 1.2.17](https://www.apache.org/dyn/closer.cgi/logging/log4j/1.2.17/log4j-1.2.17.zip)

🔧 **Jar File**:  
log4j-<version>.jar (can be added via Maven)

### ✅ ****Advantages of Log4j****

#### 1. **Categorized Logging Levels**

Helps filter and prioritize logs:

| **Level** | **Description** |
| --- | --- |
| DEBUG | Normal flow of code (e.g., method start/end) |
| INFO | Important flow confirmation (e.g., DB connected, login success) |
| WARN | Suspicious but not fatal (e.g., deprecated API used) |
| ERROR | Known exceptions (e.g., SQLException) |
| FATAL | Unknown critical exceptions (e.g., Exception, Throwable) |

#### 2. **Multiple Destinations for Log Messages**

* Console
* File
* Database
* Mail Server
* Remote Server

#### 3. **Customizable Formats**

Use various layouts:

* PatternLayout
* HTMLLayout
* XMLLayout

#### 4. **Log Filtering**

Use filters to extract specific log levels:

ALL < DEBUG < INFO < WARN < ERROR < FATAL < OFF

#### 5. **External Configuration**

* Can configure Log4j through:
  + .properties files
  + .xml files

#### 6. **Asynchronous Logging**

* Logs can be written in a separate thread — does not block main execution.

#### 7. **Industry Standard**

* Widely adopted in both legacy and modern Java applications.

### 📂 ****Common Logging Practices****

* Maintain **two log files** per application:
  1. **Common Log File**: Contains all logs (DEBUG to FATAL)
  2. **Exception Log File**: Contains only ERROR and FATAL (critical for postmortem/debugging)

### 🧪 Bug vs Issue (in Testing)

| **Term** | **Meaning** |
| --- | --- |
| **Bug** | Code exists, but it doesn't behave as expected (wrong logic) e.g.: Clicking Home redirects to About Us |
| **Issue** | Missing functionality or incomplete implementation |

### 1. 🧱 ****Logger Object****

🔹 **Purpose**:  
Controls logging for a particular Java class.

🔹 **Creation**:

Logger logger = Logger.getLogger(CurrentClassName.class);

**Example**:

java

CopyEdit

Logger logger = Logger.getLogger(BankAppProject.class);

🔹 **Logging Methods by Priority**:

logger.debug("Debug message");

logger.info("Info message");

logger.warn("Warning message");

logger.error("Error message");

logger.fatal("Fatal error message");

🔹 **Set Logger Level**:

logger.setLevel(Level.DEBUG);

* If **not set**, default level is DEBUG.
* Controls which log messages are actually recorded based on the minimum level set.

🔹 **Logger Configuration Sources**:

* Hardcoded in Java code
* External .properties or .xml configuration files

🔹 **Relationships**:

* Logger object is **associated with**:
  + **Appender** (to decide destination)
  + **Layout** (to format the message)

### 2. 🧾 ****Appender Object****

🔹 **Purpose**:  
Determines **where** the log messages should be written (i.e., the destination).

🔹 **Examples of Appenders**:

| **Appender Type** | **Description** |
| --- | --- |
| ConsoleAppender | Writes logs to console (System.out) |
| FileAppender | Writes logs to a specific file |
| RollingFileAppender | Writes logs to file, rolls over after size threshold |
| DailyRollingFileAppender | Rolls over daily |
| JDBCAppender | Writes logs to a database |
| SMTPAppender | Sends logs via email |

🔹 **Hierarchy**:  
All appender classes implement:

org.apache.log4j.Appender

### 3. 🎨 ****Layout Object****

🔹 **Purpose**:  
Controls the **format** of the log messages before they are handed over to the Appender.

🔹 **Examples of Layouts**:

| **Layout Type** | **Description** |
| --- | --- |
| SimpleLayout | Basic format: level and message |
| PatternLayout | Flexible and customizable format (most used) |
| HTMLLayout | Outputs logs in HTML table format |
| XMLLayout | Outputs logs in XML format |

🔹 **Hierarchy**:  
All layout classes extend:

org.apache.log4j.Layout

## 📌 Summary Diagram (Simplified Flow)

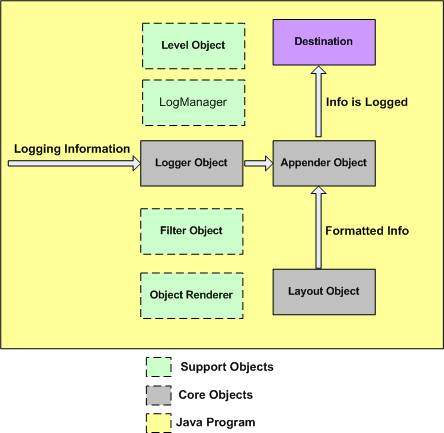
Logger (controls log creation)

|

|---> Appender (decides destination: console, file, DB, etc.)

|

---> Layout (decides format: text, XML, HTML)



## 🏗️ **Log4j Architecture Overview**

The Log4j framework consists of **core objects** and **support objects** that work together to enable flexible and powerful logging in Java applications.

### 📦 ****Core Objects (Gray Boxes)****

### 🔷 ****Logger Object****

* + Main component that receives logging information from your Java program.
  + Responsible for checking the logging level and delegating messages to the appropriate Appender.
  + Created per Java class using:

Logger logger = Logger.getLogger(MyClass.class);

### 📤 ****Appender Object****

* + Decides **where** to send the log output (file, console, database, etc.).
  + Logger passes accepted log messages to Appender.

### 🎨 ****Layout Object****

* + Formats the log message before it's written by the Appender.
  + Common layouts: PatternLayout, SimpleLayout, HTMLLayout, etc.

### 🧰 ****Support Objects (Green Boxes)****

### 🏷️ ****Level Object****

* + Defines log severity levels: DEBUG, INFO, WARN, ERROR, FATAL.
  + Logger uses this to determine whether to log a message.

### 🧠 ****LogManager****

* + Manages Logger instances.
  + Responsible for configuration loading and managing the logger hierarchy.

### 🔍 ****Filter Object****

* + Optional object to add additional filtering logic.
  + Works inside the Appender to decide whether a particular log message should be processed.

### 🖼️ ****ObjectRenderer****

* + Converts complex objects into a string form for logging.

### 🎯 ****Destination (Purple Box)****

* Final target where the formatted log message is written.
* Examples:
  + File system (e.g., info.log)
  + Console
  + Database
  + Email server
  + Remote socket server

### 🟨 ****Java Program (Yellow Background)****

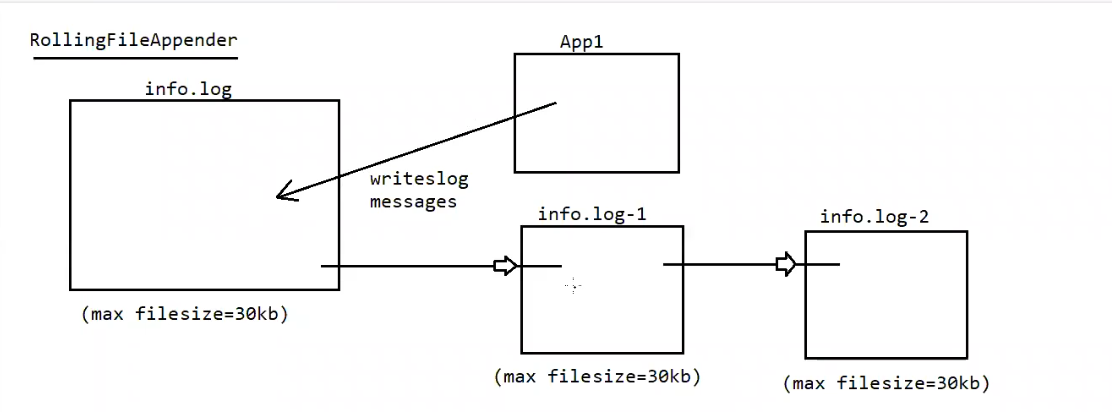
* Your application code sends **logging information** to the Logger object.

### 🔁 ****Log Flow Summary****

Java Program → Logger → (check Level) → Appender → Layout → Destination

↳ (uses Filter, Renderer)

Eg: Log4JDemo



### 🔁 ****RollingFileAppender – How It Works****

#### 📌 Purpose:

RollingFileAppender is used when you want to limit the size of your log file and automatically **roll over** to a new file when the limit is reached.

### 📂 ****Flow Explained from the Diagram****

1. **Active Log File: info.log**
   * This is the **current log file** where the application (App1) writes log messages.
   * It has a **maximum file size limit**, e.g., 30KB.
2. **When info.log exceeds 30KB**:
   * Log4j **renames** info.log to info.log-1.
   * A **new empty** info.log is created for fresh log entries.
3. **If info.log exceeds 30KB again**:
   * info.log-1 is renamed to info.log-2.
   * The current info.log is renamed to info.log-1.
   * A new info.log is created again.
4. This process **continues** for further rollovers, up to a configured **maximum backup index** (e.g., MaxBackupIndex=5).

### ⚙️ ****Log4j Properties Configuration Example****

Here’s how you would configure RollingFileAppender in log4j.properties:

properties

log4j.rootLogger=DEBUG, R

log4j.appender.R=org.apache.log4j.RollingFileAppender

log4j.appender.R.File=info.log

log4j.appender.R.MaxFileSize=30KB

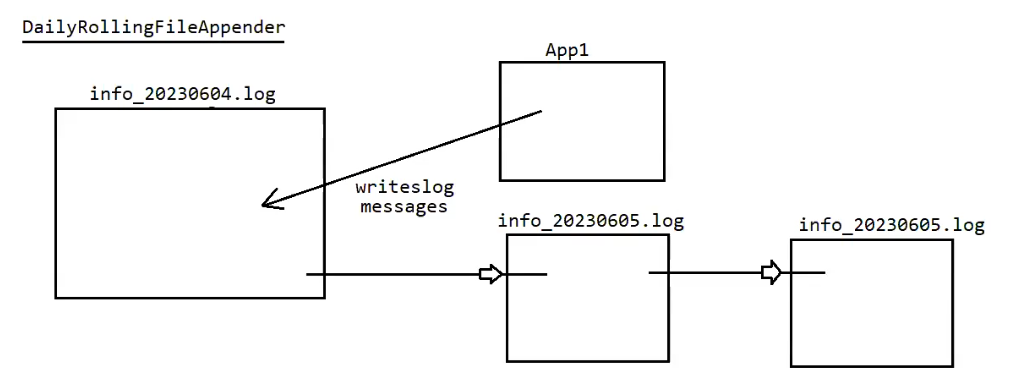
log4j.appender.R.MaxBackupIndex=5

log4j.appender.R.layout=org.apache.log4j.PatternLayout

log4j.appender.R.layout.ConversionPattern=%d [%t] %-5p %c - %m%n

### 📘 Summary

| **Component** | **Description** |
| --- | --- |
| info.log | Main active log file |
| info.log-1 | First backup when info.log is full |
| info.log-2 | Second backup (older logs) |
| MaxFileSize | Triggers rollover when size limit is hit |
| MaxBackupIndex | How many backup files to maintain |



## 📆 **DailyRollingFileAppender – How It Works**

### ✅ ****Purpose****:

* Automatically rolls over the log file based on **time intervals** rather than size.
* Ideal for separating logs by **day, hour, minute, week, or month**.

### 📂 ****Flow Explanation from the Diagram****

1. **Active Log File: info\_20230604.log**
   * The application (App1) writes log messages to this file on **June 4, 2023**.
   * File name includes the date as a **timestamp suffix**.
2. **Next Day: June 5, 2023**
   * Log4j rolls over the log file.
   * info\_20230604.log is **archived**.
   * A new file info\_20230605.log is created for the new day’s logs.
3. **Following Day: June 6, 2023**
   * Same pattern continues: new file info\_20230606.log is created.

### 🛠️ ****Log4j Properties Configuration Example****

properties

log4j.rootLogger=DEBUG, D

log4j.appender.D=org.apache.log4j.DailyRollingFileAppender

log4j.appender.D.File=info.log

log4j.appender.D.DatePattern='\_'yyyyMMdd'.log'

log4j.appender.D.layout=org.apache.log4j.PatternLayout

log4j.appender.D.layout.ConversionPattern=%d [%t] %-5p %c - %m%n

### 📘 ****DatePattern Options**** for Rolling:

| **Pattern** | **Rolls Over...** | **Example Filename** |
| --- | --- | --- |
| '.'yyyy-MM-dd | Daily | info.2025-07-03 |
| '.'yyyy-MM-dd-HH | Hourly | info.2025-07-03-15 |
| '.'yyyy-MM | Monthly | info.2025-07 |
| '.'yyyy-ww | Weekly | info.2025-27 (week 27) |
| '.'yyyy-MM-dd-HH-mm | Minutely | info.2025-07-03-15-30 |

### 🧠 ****Fun Fact****:

🔸 **Apache Tomcat server** internally uses DailyRollingFileAppender to maintain logs (e.g., catalina.<date>.log).

### 🆚 RollingFileAppender vs DailyRollingFileAppender

| **Feature** | **RollingFileAppender** | **DailyRollingFileAppender** |
| --- | --- | --- |
| Roll Trigger | File size exceeds threshold | Time interval reached (daily etc.) |
| Naming Format | info.log, info.log.1, etc | info\_yyyyMMdd.log, etc |
| Good For | High-frequency logs | Time-based analysis |