

## 🏗️ **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 in Log4j (Explained in Detail)

### 1. 🏷️ ****Level Object****

#### ✅ What It Is:

* This object defines the **importance level** of a log message.
* It helps the Logger decide **whether to log a message or ignore it**.

#### ✅ Common Log Levels (From Low to High):

DEBUG < INFO < WARN < ERROR < FATAL

#### ✅ How It Works:

* If you set the logger level to INFO, then:
  + DEBUG messages will be ignored (too detailed)
  + Only INFO, WARN, ERROR, and FATAL will be logged

#### ✅ Example:

logger.setLevel(Level.INFO);

logger.debug("This won't be logged");

logger.info("This will be logged");

📌 **Simple Rule**: If the log message level is equal to or higher than the logger level, it will be printed.

2. LogManager

### ✅ ****What It Is:****

* LogManager is a **core utility class** in Log4j.
* It is responsible for **creating**, **configuring**, and **managing all Logger objects** used across your Java application.

### ✅ ****What It Does:****

#### 1. **Loads Configuration Automatically**

* It reads logging configurations from:
  + A .properties file (log4j.properties) or
  + An .xml file (log4j.xml)
* These files tell Log4j **how logging should behave** (log levels, formats, appenders, etc.).

#### 2. **Manages Logger Instances**

* Instead of creating loggers manually with new, you use:

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

* LogManager internally checks if a logger for MyClass already exists:
  + If yes, it returns the existing one.
  + If no, it creates a new one, stores it, and returns it.

📌 This avoids creating **duplicate loggers** for the same class.  
📌 It improves **performance and memory usage**.

### ✅ 3. ****Ensures Centralized Logger Control****

🔸 This means:  
All the loggers used in your project are **controlled, created, and configured from one central place**, using **LogManager**.

### 💡 Why is this useful?

#### 🧩 Problem without LogManager:

If you manually create loggers using new Logger(), you will:

* Have **too many logger objects**
* Lose track of them
* Find it hard to **update or stop** logging across the app

### ✅ What LogManager does:

1. ✅ **Central point of control**
   * You don’t manually create loggers.
   * You use LogManager.getLogger() and get a logger with all settings **already applied**.
2. ✅ **Reads config automatically**
   * It reads settings from files like:
     + log4j.properties
     + log4j.xml
   * So you **don’t need to set levels, formats, or destinations** in every class.
3. ✅ **Applies same config to all loggers**
   * Example: If your config file says "log everything to file", **all loggers** in your app will follow that.
4. ✅ **You can change log settings from one place**
   * Want to turn off debug logs? Just change it once in the config file. All loggers will follow.
   * Want to send logs to a new file? Again, just one change needed.

### ✅ Example:

Let’s say you have 100 classes in your app, like:

* EmployeeService.java
* CustomerService.java
* OrderService.java
* and many more...

If you use:

Logger logger = LogManager.getLogger(EmployeeService.class);

* This logger:
  + Is **created once**
  + Is **managed by LogManager**
  + Automatically **follows** the settings in your config file (log level, destination, format)

You can now use:

logger.info("Service started...");

➡ You didn’t set log level or destination in your code.  
➡ LogManager applied it from the central config.

### 🔒 Bonus: Central Shutdown

* If your application needs to stop logging (like during shutdown or for debugging), you can **shut down all loggers at once** using LogManager.

## 🔍 **3. Filter Object – Explained Simply**

### ✅ ****What is a Filter Object?****

* A **Filter** is like a **gatekeeper** for your log messages.
* It **checks each message** before it gets saved to a file or shown on the screen.
* If the message **passes your rule**, it gets logged.
* If not, it is **skipped**.

### ✅ ****Where is it used?****

* A Filter **lives inside the Appender**.
* The Appender decides **where to log** (file, console, DB), and the Filter decides **whether to log**.

### ✅ ****Why do we need it?****

* Sometimes, you want more **fine control** than just log levels.
* For example:
  + You want to **log only errors from a specific class**.
  + Or you want to **skip logs during maintenance hours**.
  + Or you want to **log only messages that have a specific keyword** (like "ALERT" or "FAILURE").

### ✅ ****What does it do exactly?****

The Filter checks each log message using your custom logic.

If it passes the filter → ✅ it goes to the Appender.  
If it fails the filter → ❌ it is ignored (not logged).

### ✅ ****Real-Life Example (Simple)****

Let’s say you only want to log messages **that contain the word "ERROR"**.

You can write a simple filter like this:

if (message.contains("ERROR")) {

// Allow logging

} else {

// Skip logging

}

Now, even if your code has 100 log statements, **only the messages with "ERROR"** will be saved.

### ✅ ****Another Example (Advanced but Easy)****

You have an app with two modules:

* payment
* inventory

You only want to log messages from payment module:

if (className.contains("payment")) {

log it;

} else {

skip it;

}

### 4. 🖼️ ****ObjectRenderer****

#### ✅ What It Is:

* This object helps **convert complex Java objects** (like User, Order, etc.) into a **string format** that can be logged.

#### ✅ Why It's Needed:

* By default, Java objects may print something like User@1a2b3c.
* That’s not helpful in logs.
* ObjectRenderer lets you **customize** what gets printed.

#### ✅ What It Does:

* It **extracts useful fields** from the object and formats them as a string.

#### ✅ Example:

Let’s say you have a class:

class User {

String name;

String email;

}

Normally, logging user may give:

User@5a07e868

With ObjectRenderer, you can define:

UserRenderer: name=John, email=john@example.com

#### ✅ Real Usage:

You write a custom class that extends ObjectRenderer and define how to stringify the object. Then you register it in the Log4j config file.

### 🎯 ****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)****

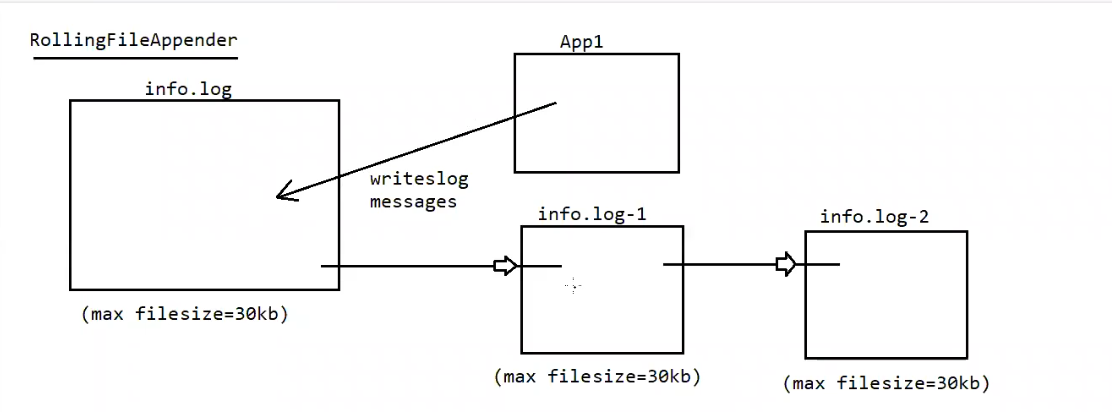
* Your application code sends **logging information** to the Logger object.

### 🔁 ****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

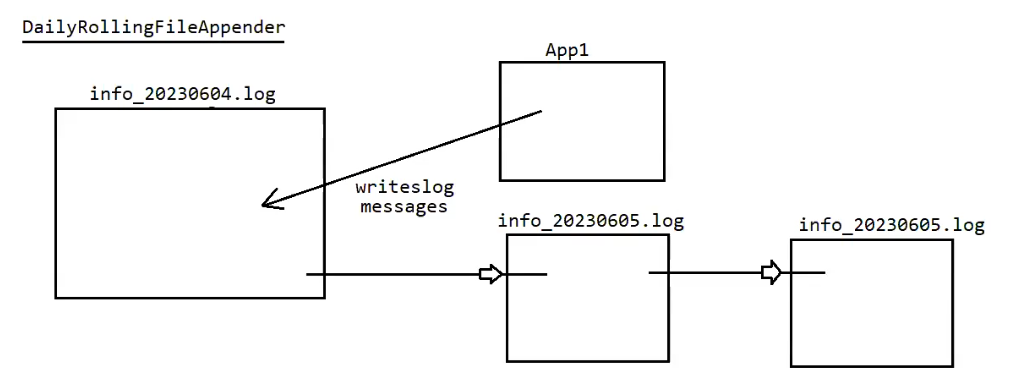
log4j.appender.R.MaxBackupIndex=5

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 |