Java logging introduction

2. Loggers, Logging Types, Logging Configuration

now we go level by level growth like a game on this java topic, provide coding example with its explanation, real-world use case example,neccessary hinds, points to remember,usual mistakes,etc.

start lvl1

Great 👍 Let's start **Level 1** of our "Java Logging Game"!
We'll begin from the basics and slowly build up to advanced logging configurations step by step.

🎮 Level 1: Introduction to Logging in Java

📌 What is Logging?

Logging is the process of **recording information** about the execution of a program.
Think of it as a **black box recorder** for your application 🚀.
It helps developers:

- Debug issues (trace the cause of an error)
- Monitor application behavior (performance, requests, errors)
- Audit user activity (important in banking, healthcare apps)
- Replace `System.out.println()` with a structured and professional way.

📜 Example: Using `System.out.println()` (Old Way)

```java
public class OldWayLogging {
    public static void main(String[] args) {
        System.out.println("Application started...");

        int x = 10, y = 0;
        try {
            int result = x / y;
            System.out.println("Result: " + result);
        } catch (Exception e) {
            System.out.println("Error occurred: " + e.getMessage());
        }

        System.out.println("Application finished...");
    }
}
```

🔴 Problems with `System.out.println()`:

- Hard to filter (prints everything).
- No logging levels (debug, info, error).
- Not configurable (cannot write to file easily).
- Clutters production logs.

📜 Example: Using `java.util.logging.Logger` (Better Way)

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

public class LoggerDemo {
    private static final Logger logger = Logger.getLogger(LoggerDemo.class.getName());

    public static void main(String[] args) {
        logger.info("Application started...");

        int x = 10, y = 0;
        try {
            int result = x / y;
            logger.info("Result: " + result);
        } catch (Exception e) {
            logger.severe("Error occurred: " + e.getMessage());
        }

        logger.info("Application finished...");
    }
}
```

✅ Benefits:

- Provides different logging levels (**`INFO`**, **`WARNING`**, **`SEVERE`**).
- Easily configurable (console, file, XML, JSON outputs).
- Better control for debugging and production.

⚙️ Logging Levels in Java

From **lowest** → **highest** importance:

1. **`SEVERE`** → Serious failure (e.g., database down, NullPointerException).
2. **`WARNING`** → Potential problem (e.g., deprecated API usage).
3. **`INFO`** → General information (application started, user logged in).
4. **`CONFIG`** → Configuration messages.
5. **`FINE`** → Debugging details.
6. **`FINER`** → More detailed debugging.
7. **`FINEST`** → Most detailed tracing.

🎯 Real-World Use Case

Imagine an **online banking application**:

- **INFO** → “User John logged in at 10:05 AM”
- **WARNING** → “Multiple failed login attempts for user John”
- **SEVERE** → “Database connection lost!”

This way, system admins can filter **only SEVERE logs** when investigating outages, instead of searching through thousands of print statements.

⚡ Hints & Points to Remember

- ✅ Always use logging instead of `System.out.println()`.
- ✅ Choose the right logging level (don’t log everything as `SEVERE`).
- ✅ Logs can be written to **files, databases, monitoring tools**.
- ❌ Don’t log sensitive information (passwords, credit card numbers).
- ❌ Avoid excessive logging in production (it may slow down app).

🚩 Usual Mistakes

1. Using `System.out.println()` in production code.
2. Logging too much data (hard to find useful info).
3. Not differentiating between `INFO`, `WARNING`, and `SEVERE`.
4. Forgetting to configure logging properly (default may not store logs).

✅ Level 1 Complete 🎉
You now understand:

- Why logging is better than `System.out.println()`.
- Basic use of `java.util.logging.Logger`.
- Logging levels and real-world use cases.

👉 Shall we move to **Level 2**, where we explore **Logging Types (Console vs File Logging)** with examples?

next

Perfect 🚀 let’s move to **Level 2** of our Logging Game.

🎮 Level 2: Logging Types (Console vs File Logging)

Now that you know the **basics of logging**, let’s explore the **types of logging** and how to configure them.

📌 Types of Logging

1. **Console Logging** → Logs appear directly in the terminal (default).
2. **File Logging** → Logs are saved into files for later analysis.
3. **Remote/Database Logging** → Logs are sent to monitoring tools (like ELK stack, Splunk).
4. **Rolling Logging** → Logs rotate automatically after reaching a size limit.

For **Level 2**, we'll focus on **Console** and **File Logging**.

🖥️ Example 1: Console Logging (Default)

By default, Java's `Logger` writes to the console.

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

public class ConsoleLoggingDemo {
    private static final Logger logger = Logger.getLogger(ConsoleLoggingDemo.class.getName());

    public static void main(String[] args) {
        logger.info("Application started...");
        logger.warning("This is a warning message!");
        logger.severe("Something went wrong!");
    }
}
```

👉 Output on Console:

```yaml
Sep 25, 2025 10:00:00 AM ConsoleLoggingDemo main
INFO: Application started...
Sep 25, 2025 10:00:00 AM ConsoleLoggingDemo main
WARNING: This is a warning message!
Sep 25, 2025 10:00:00 AM ConsoleLoggingDemo main
SEVERE: Something went wrong!
```

📁 Example 2: File Logging

We can configure logging to **write logs into a file**.

```java
import java.io.IOException;
import java.util.logging.*;

public class FileLoggingDemo {
    private static final Logger logger = Logger.getLogger(FileLoggingDemo.class.getName());

    public static void main(String[] args) {
        try {
            // Create a FileHandler that writes log messages to app.log
            FileHandler fileHandler = new FileHandler("app.log", true); // true = append mode
            fileHandler.setFormatter(new SimpleFormatter()); // Format logs
            logger.addHandler(fileHandler);

            logger.info("Application started...");
            logger.warning("This warning will also go to the log file!");
            logger.severe("Severe error occurred!");

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

👉 After running, check `app.log` file:

```yaml
Sep 25, 2025 10:05:00 AM FileLoggingDemo main
INFO: Application started...
Sep 25, 2025 10:05:00 AM FileLoggingDemo main
WARNING: This warning will also go to the log file!
Sep 25, 2025 10:05:00 AM FileLoggingDemo main
SEVERE: Severe error occurred!
```

🎯 Real-World Use Case

Imagine an **e-commerce website**:

- Developers see console logs while testing locally.
- In production, logs are stored in **files**:
  - `info.log` → user activities (logins, purchases).
  - `error.log` → only errors and failures.

When a customer reports “Payment failed,” the support team can check `error.log` instead of reading through thousands of debug lines.

⚡ Hints & Points to Remember

- ✅ Use **console logging** for local debugging.
- ✅ Use **file logging** for production monitoring.
- ✅ Always use `append = true` for `FileHandler` (so logs aren’t overwritten).
- ❌ Don’t keep logs only in memory; write them to files or monitoring systems.
- ❌ Avoid logging sensitive information (PIN, passwords).

🚩 Usual Mistakes

1. Forgetting to close or configure `FileHandler` → file may get locked.
2. Overwriting logs (not using append mode).

3. Logging too much (huge log files consume disk space).
4. Using only console logs in production (you lose logs when server restarts).

✅ Level 2 Complete 🎉
You now understand:

- Console vs File Logging
- How to configure logging into a file
- Real-world e-commerce logging scenario

👉 Shall we move to **Level 3**, where we'll cover **Logging Configuration (Handlers, Formatters, Log Levels)** with more control?

next

Awesome ⚡ Let's enter **Level 3** of our Logging Game.
Now we go deeper into **Logging Configuration** — how to **control what logs appear, where they go, and how they look**.

🎮 Level 3: Logging Configuration (Handlers, Formatters, Log Levels)

📌 Key Components in Java Logging

1. **Logger** → Main object you use (`Logger.getLogger()`).
2. **Handler** → Decides where logs go (Console, File, Socket, DB).
3. **Formatter** → Decides how logs look (plain text, XML, JSON).
4. **Level** → Filters logs (INFO, WARNING, SEVERE, etc).

⚙️ Example 1: Custom Handler + Level Filtering

```java
import java.io.IOException;
import java.util.logging.*;

public class LoggingConfigDemo {
    private static final Logger logger = Logger.getLogger(LoggingConfigDemo.class.getName());

    public static void main(String[] args) {
        try {
            // Remove default console handler
            LogManager.getLogManager().reset();
            // File Handler
            FileHandler fileHandler = new FileHandler("app.log", true);
            fileHandler.setLevel(Level.ALL); // Accept all logs
            fileHandler.setFormatter(new SimpleFormatter());

            // Console Handler
            ConsoleHandler consoleHandler = new ConsoleHandler();
            consoleHandler.setLevel(Level.WARNING); // Only warnings & errors go to console

            // Add handlers to logger
            logger.addHandler(fileHandler);
            logger.addHandler(consoleHandler);

            // Log messages with different levels
            logger.info("This is INFO log");
            logger.warning("This is WARNING log");
            logger.severe("This is SEVERE log");

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

👉 Behavior:

- **Console** shows only **`WARNING`** and **`SEVERE`**.
- **File (`app.log`)** contains everything (**`INFO`**, **`WARNING`**, **`SEVERE`**).

⚙️ Example 2: Custom Formatter

By default, logs look like this:

```pgsql
Sep 25, 2025 10:30:00 AM LoggingConfigDemo main
INFO: This is INFO log
```

We can make them **cleaner** 👇

```java
class MyFormatter extends Formatter {
    @Override
    public String format(LogRecord record) {
        return "[" + record.getLevel() + "] " + record.getMessage() + "\n";
    }
}
```

Use it in code:

```java
fileHandler.setFormatter(new MyFormatter());
```

👉 Output in **`app.log`**:

```pgsql
[INFO] This is INFO log
[WARNING] This is WARNING log
[SEVERE] This is SEVERE log
```

🎯 Real-World Use Case

In a **hospital management system**:

- Console logs: Only show warnings/errors (to avoid clutter for admins).
- File logs: Store full history (including info/debug) for auditing.
- Formatter: Custom JSON logs to integrate with monitoring tools (Splunk, ELK).

⚡ Hints & Points to Remember

- ✅ Use **different handlers** for different destinations (console, file, monitoring).
- ✅ Set **log levels carefully** (don't spam console in production).
- ✅ Use **custom formatters** for structured logs (JSON/XML).
- ❌ Don't keep default format if your team needs structured logs.
- ❌ Don't forget `LogManager.getLogManager().reset();` to avoid duplicate logs.

🚩 Usual Mistakes

1. Forgetting to remove default handler → logs appear **twice**.
2. Not setting log level properly → either too much noise or missing critical logs.
3. Using same format for console and file → console should be short, file can be detailed.
4. Writing logs only to console → you lose history after restart.

✅ Level 3 Complete 🎉
You now know:

- Handlers (Console/File)
- Formatters (custom log style)
- Log Levels (filtering)

👉 Do you want me to take you to **Level 4**, where we'll explore **External Logging Frameworks (Log4j2, SLF4J, Logback)** and how they're used in **real enterprise projects**?

next

Awesome! 🚀 Let's move to **Level 4**: **External Logging Frameworks** – this is where professional Java applications really shine.

🎮 Level 4: External Logging Frameworks (Log4j2, SLF4J, Logback)

While `java.util.logging` is okay for simple projects, **real-world enterprise apps** use **more powerful logging frameworks** like **Log4j2** and **SLF4J + Logback** for advanced features.

📌 Why Use External Logging Frameworks?

1. More **flexible configuration** (XML, YAML, properties files).
2. Better **performance** (asynchronous logging).
3. Supports **rolling files, compression, filtering, multiple appenders**.
4. Industry standard → integrates with **Spring Boot**, **microservices**, **monitoring tools**.

⚡ Example 1: Log4j2 Basic Setup

Step 1: Add Maven Dependency

xml

```xml
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.20.0</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.20.0</version>
</dependency>
```

Step 2: Create `log4j2.xml` Configuration

xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="[%d{HH:mm:ss}] [%level] %msg%n"/>
        </Console>
        <File name="FileLogger" fileName="app.log">
            <PatternLayout pattern="[%d{yyyy-MM-dd HH:mm:ss}] [%level] %msg%n"/>
        </File>
    </Appenders>
    <Loggers>
        <Root level="info">
            <AppenderRef ref="Console"/>
            <AppenderRef ref="FileLogger"/>
        </Root>
    </Loggers>
</Configuration>
```

Step 3: Use Logger in Code

```java
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class Log4jDemo {
    private static final Logger logger = LogManager.getLogger(Log4jDemo.class);

    public static void main(String[] args) {
        logger.info("Application started");
        logger.warn("This is a warning");
        logger.error("This is an error");
    }
}
```

✅ Features:

- Logs appear in **console** and **file** automatically.
- Configurable without changing code.
- Supports **pattern layouts, rolling files, async logging**.

⚡ Example 2: SLF4J + Logback (Spring Boot Standard)

Step 1: Maven Dependencies

```xml
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>2.0.9</version>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.4.11</version>
</dependency>
```

Step 2: `logback.xml` Configuration

```xml
<configuration>
    <appender name="Console" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>[%d{HH:mm:ss}] [%level] %msg%n</pattern>
        </encoder>
    </appender>

    <appender name="File" class="ch.qos.logback.core.FileAppender">
        <file>app.log</file>
        <encoder>
            <pattern>[%d{yyyy-MM-dd HH:mm:ss}] [%level] %msg%n</pattern>
        </encoder>
    </appender>
    <root level="info">
        <appender-ref ref="Console"/>
        <appender-ref ref="File"/>
    </root>
</configuration>
```

Step 3: Use Logger in Code

```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class LogbackDemo {
    private static final Logger logger = LoggerFactory.getLogger(LogbackDemo.class);

    public static void main(String[] args) {
        logger.info("Application started");
        logger.debug("Debugging info");
        logger.error("An error occurred");
    }
}
```

✅ Advantages of SLF4J:

- Acts as a **logging facade** → you can switch backend (Log4j2, Logback) easily.
- Supports **parameterized logging** → faster than string concatenation:

```java
logger.info("User {} logged in at {}", username, time);
```

🎯 Real-World Use Case

Imagine a **microservices architecture**:

- Service A logs to **file**.
- Service B logs to **database** via Logback DB appender.
- Centralized monitoring (ELK stack) reads all logs in JSON format.
- Errors trigger alerts automatically (Slack, email).

⚡ Hints & Points to Remember

- ✅ Use external frameworks for **production-grade applications**.
- ✅ Always configure **separate appenders** for console, file, and remote monitoring.
- ✅ Use parameterized logs to **improve performance**.
- ❌ Don't log sensitive info (PII, passwords) to centralized logs.
- ❌ Avoid using multiple logging frameworks in one project (leads to conflicts).

🚩 Usual Mistakes

1. Forgetting to add proper dependency → logger won't work.
2. Not using **rolling files** → logs grow indefinitely.
3. Logging sensitive data in centralized logs → compliance issues.
4. Using blocking loggers in high-performance apps → slows system down.

✅ Level 4 Complete 🎉
You now understand:

- Why enterprises use Log4j2 / SLF4J + Logback.
- How to configure appenders, formatters, and log levels.
- Parameterized logging and real-world microservices scenario.

If you want, **Level 5** can cover **Advanced Logging Techniques**:

- **Asynchronous logging**
- **Rolling & archived logs**
- **Custom filters**
- **Performance tuning**

Do you want me to proceed to Level 5?

i want a program to complete undestand the use of logger

Perfect! Let's create a **complete Java program** that demonstrates most of the **practical uses of logging**:

- Logging **different levels** (`INFO`, `DEBUG`, `WARNING`, `ERROR`)
- Logging to **console** and **file**
- Using **custom formatter**
- Catching **exceptions**
- Parameterized logging

We'll use `java.util.logging` for simplicity, but the concepts are the same for Log4j2 or Logback.

Complete Logger Program

java

```java
import java.io.IOException;
import java.util.logging.*;

// Custom Formatter to make logs cleaner
class MyFormatter extends Formatter {
    @Override
    public String format(LogRecord record) {
        return String.format("[%s] %s: %s%n",
                record.getLevel(),
                record.getLoggerName(),
                record.getMessage());
    }
}

public class CompleteLoggerExample {

    private static final Logger logger =
Logger.getLogger(CompleteLoggerExample.class.getName());

    public static void main(String[] args) {
        try {
            // Reset default handlers
            LogManager.getLogManager().reset();

            // Console Handler
            ConsoleHandler consoleHandler = new ConsoleHandler();
            consoleHandler.setLevel(Level.WARNING); // only warnings and errors
            consoleHandler.setFormatter(new MyFormatter());
            logger.addHandler(consoleHandler);

            // File Handler
            FileHandler fileHandler = new FileHandler("application.log", true); // append mode
            fileHandler.setLevel(Level.ALL); // all logs
            fileHandler.setFormatter(new MyFormatter());
            logger.addHandler(fileHandler);

            // Set global log level
            logger.setLevel(Level.ALL);

            // Application Simulation
            logger.info("Application started");

            String[] users = {"Alice", "Bob", "Charlie"};
            for (String user : users) {
                logger.fine("Processing user: " + user); // debug info
                if (user.equals("Bob")) {
                    logger.warning("User Bob has incomplete profile");
                }
            }

            // Simulate an exception
            try {
                int result = 10 / 0; // will throw ArithmeticException
            } catch (Exception e) {
                logger.severe("Exception occurred: " + e.getMessage());
            }

            logger.info("Application finished");

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✅ What This Program Demonstrates

1. **Multiple Handlers**

  - **ConsoleHandler** → shows only `WARNING` and `SEVERE`.
  - **FileHandler** → logs **all levels** to `application.log`.
2. **Custom Formatter** → `[LEVEL] LoggerName: Message`
3. **Logging Levels**
   - `INFO` → general info
   - `FINE` → debug
   - `WARNING` → potential issues
   - `SEVERE` → critical issues
4. **Exception Logging**
   - Exceptions are logged with `severe` level.
5. **Appending to File**
   - `true` ensures logs are appended instead of overwritten.

🔥 Sample Output

Console:

```pgsql
[WARNING] CompleteLoggerExample: User Bob has incomplete profile
[SEVERE] CompleteLoggerExample: Exception occurred: / by zero
```

application.log:

```pgsql
[INFO] CompleteLoggerExample: Application started
[FINE] CompleteLoggerExample: Processing user: Alice
[FINE] CompleteLoggerExample: Processing user: Bob
[WARNING] CompleteLoggerExample: User Bob has incomplete profile
[FINE] CompleteLoggerExample: Processing user: Charlie
[SEVERE] CompleteLoggerExample: Exception occurred: / by zero
[INFO] CompleteLoggerExample: Application finished
```

🎯 Real-World Use Case

Think of this as a **user management module in a web application**:

- `INFO` → Track application start/end or user actions.
- `FINE` → Debug info for developers (only enabled in dev environment).
- `WARNING` → Suspicious or incomplete data.
- `SEVERE` → Runtime exceptions or critical failures.

This single program teaches **all core logging concepts** used in production.

If you want, I can also make an **enhanced version using Log4j2** that shows **rolling files, async logging, and parameterized logs**, which is **closer to enterprise usage**.

Do you want me to make that version too?

does which make to store log ans which prints log

Ah! Let me clarify clearly — in Java logging, **different components have different responsibilities**:

1️⃣ Logger

- The **Logger** is the **main interface** you use in your code to log messages.
- It **does not directly print or store logs**.
- It decides **what message to log** and **which handlers** should process it.

Example:

```java
Logger logger = Logger.getLogger("MyLogger");
logger.info("Application started"); // Logger generates this log
```

2️⃣ Handler / Appender

- This is what **actually decides where the log goes**.
- Types:
  1. **ConsoleHandler** → prints log to **console/terminal**
  2. **FileHandler** → writes log to a **file**
  3. **SocketHandler, DBHandler, Custom Handler** → sends logs elsewhere (network, DB, monitoring tools)

Example:

```java
ConsoleHandler ch = new ConsoleHandler(); // prints to console
FileHandler fh = new FileHandler("app.log"); // stores in file

logger.addHandler(ch);
logger.addHandler(fh);
```

So **Logger** → creates the message, **Handler** → decides where it goes.

3️⃣ Formatter

- Formatter decides **how the log looks** (timestamp, level, message).
- It doesn't print or store by itself, but formats the message for the handler.

Example:

```java
fh.setFormatter(new SimpleFormatter()); // default readable text
```

🔹 Summary Table

Component	Responsibility	Example Output
Logger	Generates/logs the message	`"Application started"`
Handler	Prints/stores log (console/file/etc.)	Console print / writes to file
Formatter	Formats the log	`[INFO] MyLogger: Application started`

So in our **complete logger program**:

- `logger.info("Application started")` → **Logger generates message**
- `ConsoleHandler` → prints WARNING/SEVERE to **console**
- `FileHandler` → writes **all logs** to `application.log`
- `MyFormatter` → formats logs for **readability**

If you want, I can draw a **diagram showing the Logger → Handler → Formatter flow**, which makes this **super easy to visualize**.

Do you want me to make that diagram?

logger.info("Application finished");

does this makes, if the level is console it prints, if the level is file it store

Exactly! ✅ Let me break it down clearly:

1️⃣ How `logger.info("Application finished")` works

1. **Logger receives the log request**
   - You call `logger.info("Application finished")`.
   - Logger checks its **level** and the **level of its handlers**.
2. **Handlers decide what to do**

- Each **Handler** (ConsoleHandler, FileHandler, etc.) has its own **level filter**.
- If the log's level meets or exceeds the handler's level:
    - **ConsoleHandler** → prints to console.
    - **FileHandler** → writes to file.
- If it doesn't meet the handler's level → the handler ignores it.

2️⃣ Example from our previous program

```java
ConsoleHandler consoleHandler = new ConsoleHandler();
consoleHandler.setLevel(Level.WARNING); // only warnings & errors go to console
logger.addHandler(consoleHandler);

FileHandler fileHandler = new FileHandler("app.log", true);
fileHandler.setLevel(Level.ALL); // all logs go to file
logger.addHandler(fileHandler);

logger.setLevel(Level.ALL);
logger.info("Application finished");
logger.warning("Potential issue detected");
```

✅ Behavior:

Log Statement	Console Output	File Output
`logger.info("Application finished")`	❌ Not printed (ConsoleHandler = WARNING)	✅ Stored in file
`logger.warning("Potential issue detected")`	✅ Printed	✅ Stored in file

3️⃣ Key Concept

- **Logger** → generates the message.
- **Handler** → decides **where** the log goes based on its **level**.
- **Level mismatch** → log is **ignored** by that handler.

So yes:

- If the handler is **console and level matches**, it prints.
- If the handler is **file and level matches**, it stores.

If you want, I can make a **small visual flow diagram** showing **Logger** → **Handlers** → **Console/File** with levels, which makes this **super clear**.

Do you want me to make that diagram?

example for sinlge logger for multiple objects

Perfect! Let’s explore **how a single `Logger` can be used across multiple objects** in Java. This is a common pattern in real-world apps: you don’t create a separate logger for every object, but rather **share one logger per class** or even globally.

✅ Example: Single Logger for Multiple Objects

```java
import java.util.logging.*;

class User {
    private String name;
    private static final Logger logger = Logger.getLogger(User.class.getName());

    public User(String name) {
        this.name = name;
        logger.info("User object created: " + name);
    }

    public void login() {
        logger.info(name + " logged in");
    }

    public void logout() {
        logger.info(name + " logged out");
    }
}

public class SingleLoggerExample {
    private static final Logger logger = Logger.getLogger(SingleLoggerExample.class.getName());

    public static void main(String[] args) {
        try {
            // Configure a single FileHandler for demonstration
            FileHandler fileHandler = new FileHandler("users.log", true);
            fileHandler.setFormatter(new SimpleFormatter());

            // Add the same handler to both loggers
            logger.addHandler(fileHandler);
            User.logger.addHandler(fileHandler); // Static logger inside User class

            // Create multiple objects
            User user1 = new User("Alice");
            User user2 = new User("Bob");

            // Call methods
            user1.login();
            user2.login();
            user1.logout();
            user2.logout();

            logger.info("All user activities logged successfully");

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

🔹 How This Works

1. **Single Logger per class**:
    - **`User.logger`** → shared among all **`User`** objects.
    - **`SingleLoggerExample.logger`** → for the main class.
2. **FileHandler is shared**:
    - Both loggers write to the **same `users.log` file**.
3. **Multiple objects can log to same destination**:
    - You don't need a new logger per object.
    - Keeps logs consistent and easier to manage.

🔹 Sample `users.log` Output

```
pgsql

Sep 25, 2025 12:00:00 PM User <init>
INFO: User object created: Alice
Sep 25, 2025 12:00:00 PM User <init>
INFO: User object created: Bob
Sep 25, 2025 12:00:00 PM User login
INFO: Alice logged in
Sep 25, 2025 12:00:01 PM User login
INFO: Bob logged in
Sep 25, 2025 12:00:02 PM User logout
INFO: Alice logged out
Sep 25, 2025 12:00:03 PM User logout
INFO: Bob logged out
Sep 25, 2025 12:00:04 PM SingleLoggerExample main
INFO: All user activities logged successfully
```

🔹 Key Points

- ✅ **Static Logger** inside a class is the standard practice for multiple objects.
- ✅ **Single FileHandler** can serve multiple loggers (all classes write to the same file).
- ✅ Avoid creating a new Logger for **every object** — it's inefficient.
- ✅ Loggers are **thread-safe**, so multiple objects can safely log concurrently.

If you want, I can create an **even better example with one Logger for the whole application**, where **all classes and objects share the same logger and file**, which is how enterprise apps do it.

Do you want me to do that?

why two logger

Ah! Good question — let's clarify why there seemed to be **two loggers** in that example and whether it's necessary.

1️⃣ Why we had two loggers in the previous example

```java
private static final Logger logger = Logger.getLogger(User.class.getName());
```

- This logger belongs to the **User class**.
- Every instance of **`User`** shares this **one static logger**.
- It logs all activities related to users (**`login`**, **`logout`**, object creation`).

```java
private static final Logger logger = Logger.getLogger(SingleLoggerExample.class.getName());
```

- This logger belongs to the **main class** (**`SingleLoggerExample`**).
- It logs messages related to the **application itself**, not a particular user object.

So the **two loggers serve different scopes**:

- **User.logger** → logs user-related events
- **Main class logger** → logs application-level events

2️⃣ Do we really need two loggers?

- **Not always.**
- You can use **one global logger** for the whole application, especially for small apps.
- The reason to have **different loggers per class**:
  1. Makes it easy to **filter logs by class**.
  2. Helps in **large enterprise apps** where many modules log to the same file.
  3. You can **configure log levels per class**.

3️⃣ Example: Single Logger for Whole App

```java
import java.util.logging.*;

class User {
    private String name;
    private final Logger logger; // instance uses global logger

    public User(String name, Logger globalLogger) {
        this.name = name;
        this.logger = globalLogger;
        logger.info("User object created: " + name);
    }

    public void login() {
        logger.info(name + " logged in");
    }
    public void logout() {
        logger.info(name + " logged out");
    }
}

public class SingleLoggerApp {
    public static void main(String[] args) throws Exception {
        Logger globalLogger = Logger.getLogger("AppLogger");
        FileHandler fileHandler = new FileHandler("app.log", true);
        fileHandler.setFormatter(new SimpleFormatter());
        globalLogger.addHandler(fileHandler);
        globalLogger.setLevel(Level.ALL);

        User user1 = new User("Alice", globalLogger);
        User user2 = new User("Bob", globalLogger);

        user1.login();
        user2.login();
        user1.logout();
        user2.logout();

        globalLogger.info("All activities logged using single logger");
    }
}
```

✅ Now:

- **Only one logger** (`globalLogger`) for the entire app.
- Both `User` objects and main class log through it.
- Cleaner and simpler for small/medium apps.

🔹 Key Takeaways

- Two loggers were used in the previous example to **separate concerns** (class-specific logging).
- One logger is enough if you want **all logs in one place**.
- In **enterprise apps**, having **one logger per class/module** gives better **filtering, debugging, and configuration control**.

If you want, I can draw a **diagram showing "Single Logger vs Class-specific Logger"**, which makes it super clear why one or two loggers might be used.

Do you want me to make that diagram?

explain about logger levels with example

Absolutely! Let's go **level by level** with **Logger levels** in Java, including a **practical example** so you can see how they behave.

🎮 Logger Levels in Java

Java logging has **7 standard levels**, from **least to most severe**:

Level	Purpose / Usage
`SEVERE`	Serious failures (application crash, DB down)
`WARNING`	Potential problems or risky situations
`INFO`	General runtime information (startup, shutdown, user actions)
`CONFIG`	Configuration messages (system settings, parameters)
`FINE`	Debug information (detailed steps)
`FINER`	More detailed debugging
`FINEST`	Most detailed tracing (everything)

> **Hint:** You usually configure **handlers** (Console/File) to show only certain levels, e.g., **console shows WARNING+**, file stores ALL.

📌 Example: Logger Levels in Action

```java
import java.io.IOException;
import java.util.logging.*;

public class LoggerLevelsExample {
    private static final Logger logger = Logger.getLogger(LoggerLevelsExample.class.getName());

    public static void main(String[] args) throws IOException {
        // Setup File Handler
        FileHandler fileHandler = new FileHandler("levels.log", true);
        fileHandler.setFormatter(new SimpleFormatter());
        fileHandler.setLevel(Level.ALL); // store all logs in file

        // Setup Console Handler
        ConsoleHandler consoleHandler = new ConsoleHandler();
        consoleHandler.setLevel(Level.WARNING); // show WARNING and SEVERE only
        consoleHandler.setFormatter(new SimpleFormatter());

        logger.addHandler(fileHandler);
        logger.addHandler(consoleHandler);

        // Set global logger level
        logger.setLevel(Level.ALL);

        // Logging messages at different levels
        logger.finest("FINEST - Most detailed debug");
        logger.finer("FINER - Detailed debug");
        logger.fine("FINE - Debug information");
        logger.config("CONFIG - Configuration info");
        logger.info("INFO - Application running normally");
        logger.warning("WARNING - Potential problem detected");
        logger.severe("SEVERE - Serious failure occurred");
    }
}
```

🔹 Behavior

Console Output:

```yaml
Sep 25, 2025 12:30:00 PM LoggerLevelsExample main
WARNING: WARNING - Potential problem detected
Sep 25, 2025 12:30:00 PM LoggerLevelsExample main
SEVERE: SEVERE - Serious failure occurred
```

File Output (`levels.log`):

```yaml
Sep 25, 2025 12:30:00 PM LoggerLevelsExample main
FINEST: FINEST - Most detailed debug
Sep 25, 2025 12:30:00 PM LoggerLevelsExample main
FINER: FINER - Detailed debug
Sep 25, 2025 12:30:00 PM LoggerLevelsExample main
FINE: FINE - Debug information
Sep 25, 2025 12:30:00 PM LoggerLevelsExample main
CONFIG: CONFIG - Configuration info
Sep 25, 2025 12:30:00 PM LoggerLevelsExample main
INFO: INFO - Application running normally
Sep 25, 2025 12:30:00 PM LoggerLevelsExample main
WARNING: WARNING - Potential problem detected
Sep 25, 2025 12:30:00 PM LoggerLevelsExample main
SEVERE: SEVERE - Serious failure occurred
```

🔹 Key Takeaways

1. **Logger Levels** help filter logs:
   - Handlers decide which levels to **process**.
   - Logger decides which levels to **generate**.
2. **Fine-grained logging** (`FINE`, `FINER`, `FINEST`) is mostly for **development/debugging**.
3. **Info/Warning/Severe** → usually used in **production**.
4. Console can show **only important logs**, file can store **everything**.
5. Avoid logging sensitive info at **all levels**, especially in production logs.

🔹 Analogy

- `FINEST` → A microscope (sees everything, very detailed)
- `FINE` → Normal magnifying glass (good for debugging)
- `INFO` → CCTV camera (records normal events)
- `WARNING` → Alarm sensor (something unusual)
- `SEVERE` → Fire alarm (critical problem)

If you want, I can make a **diagram showing how Logger and Handler levels filter logs**, which makes it **super easy to visualize why some logs appear in console and others in file**.

Do you want me to make that diagram?