**Part 1: Why is Logging Important?**

When you're developing, it's easy to just print information to the console: System.out.println("The value of x is: " + x);

**This is a bad practice in real applications because:**

* **It's all or nothing:** You can't turn it off without changing the code.
* **No severity:** An error message ("Database connection failed!") looks the same as an informational message ("User logged in").
* **No context:** You don't know *when* it happened, or from *which part* of the code.
* **Performance:** It can slow down your application.

A **logging framework** solves these problems by providing a robust way to record application events.

**Part 2: Understanding Logging Levels**

Logging levels are used to categorize the severity of a message. This allows you to control how much detail you want to see. If you're debugging, you might want to see everything. In production, you might only want to see errors.

The standard levels, from most to least severe, are:

* **ERROR**: A serious problem that prevented a specific operation from completing. The application can often continue running.
  + *Example:* "Failed to connect to the payment gateway."
* **WARN** (Warning): An unexpected situation or potential problem that is not a critical error. The operation still completed.
  + *Example:* "The API response took 3 seconds, which is longer than the 2-second threshold." or "Disk space is running low."
* **INFO**: High-level informational messages that highlight the progress and state of the application.
  + *Example:* "Application has started successfully." or "User with ID 123 has logged in."
* **DEBUG**: Fine-grained informational events that are most useful for debugging an application. You would typically turn this on only when investigating a specific issue.
  + *Example:* "The value of variable 'customerID' is now 456."
* **TRACE**: The most detailed level of information, providing even more fine-grained debugging data than DEBUG.
  + *Example:* "Entering method calculatePrice() with parameters A and B."

**The Golden Rule:** You configure a *minimum level*. The logger will then show messages at that level **and all levels more severe than it**.

* If you set the level to INFO, you will see INFO, WARN, and ERROR messages.
* If you set the level to WARN, you will only see WARN and ERROR messages.

**Part 3: Practical Exercise with SLF4J and Logback**

We will use **SLF4J** as our logging API (the "what") and **Logback** as the implementation (the "how"). This is the most common and recommended combination in the Java world.

**Step 1: Project Setup (Maven)**

In your pom.xml, add the following dependencies. This is similar to how you added JUnit.

XML

<dependencies>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>2.0.13</version>

</dependency>

<dependency>

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

<artifactId>logback-classic</artifactId>

<version>1.5.6</version>

<scope>runtime</scope>

</dependency>

</dependencies>

After adding these, **reload your Maven project** in your IDE.

**Step 2: Create the Logback Configuration File**

This is the most important part. You control logging **outside of your code**, in a configuration file. Create a new file named logback.xml in the src/main/resources directory.

**Path:** src/main/resources/logback.xml

XML

<configuration>

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

<encoder>

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

</encoder>

</appender>

<root level="INFO">

<appender-ref ref="STDOUT" />

</root>

</configuration>

**Step 3: Write Java Code with Logging**

Now, let's create a class that performs some operations and logs messages at different levels.

**Path:** src/main/java/com/example/UserService.java

Java

package com.example;

// Import the SLF4J classes

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class UserService {

// 1. Get a logger instance for this specific class

private static final Logger logger = LoggerFactory.getLogger(UserService.class);

public void processUser(int userId) {

// Use placeholder {} for variables - it's much more efficient!

logger.info("Starting to process user with ID: {}", userId);

logger.debug("Checking user status in the database...");

// Pretend we do some database work here

if (userId == 0) {

// This is an error condition. We log it and include the exception.

logger.error("Invalid user ID received: {}", userId, new IllegalArgumentException("User ID cannot be zero"));

return; // Stop processing

}

logger.debug("User status is OK. Now checking permissions.");

// Pretend we do more work

if (userId == 123) {

logger.warn("User with ID {} is a legacy user. Consider migrating their account.", userId);

}

logger.info("Successfully finished processing user with ID: {}", userId);

}

}

**Step 4: Run the Code and Analyze the Output**

Create a Main class to run the UserService.

**Path:** src/main/java/com/example/Main.java

Java

package com.example;

public class Main {

public static void main(String[] args) {

UserService userService = new UserService();

System.out.println("--- Running with a valid user ---");

userService.processUser(123);

System.out.println("\n--- Running with a legacy user ---");

userService.processUser(456);

System.out.println("\n--- Running with an invalid user ---");

userService.processUser(0);

}

}

**Expected Console Output:**

--- Running with a valid user ---

2025-06-29 19:15:30.123 [main] INFO com.example.UserService - Starting to process user with ID: 123

2025-06-29 19:15:30.124 [main] WARN com.example.UserService - User with ID 123 is a legacy user. Consider migrating their account.

2025-06-29 19:15:30.124 [main] INFO com.example.UserService - Successfully finished processing user with ID: 123

--- Running with a legacy user ---

2025-06-29 19:15:30.125 [main] INFO com.example.UserService - Starting to process user with ID: 456

2025-06-29 19:15:30.125 [main] INFO com.example.UserService - Successfully finished processing user with ID: 456

--- Running with an invalid user ---

2025-06-29 19:15:30.126 [main] INFO com.example.UserService - Starting to process user with ID: 0

2025-06-29 19:15:30.126 [main] ERROR com.example.UserService - Invalid user ID received: 0

java.lang.IllegalArgumentException: User ID cannot be zero

at com.example.UserService.processUser(UserService.java:18)

at com.example.Main.main(Main.java:14)

**Analysis:** Notice that the DEBUG messages ("Checking user status...") are **not** displayed. This is because our root level in logback.xml is set to INFO.

**Part 4: Your Task - Changing the Warning Level**

Now, let's see how easy it is to change the logging detail **without touching a single line of Java code.**

1. **Modify logback.xml:** Go back to your logback.xml file and change the root level from INFO to DEBUG.

XML

<root level="DEBUG">

<appender-ref ref="STDOUT" />

</root>

1. **Re-run the Main class.**

**New Console Output:**

Now you will see the DEBUG messages included!

--- Running with a valid user ---

2025-06-29 19:18:45.555 [main] INFO com.example.UserService - Starting to process user with ID: 123

2025-06-29 19:18:45.558 [main] DEBUG com.example.UserService - Checking user status in the database...

2025-06-29 19:18:45.558 [main] DEBUG com.example.UserService - User status is OK. Now checking permissions.

2025-06-29 19:18:45.558 [main] WARN com.example.UserService - User with ID 123 is a legacy user. Consider migrating their account.

2025-06-29 19:18:45.558 [main] INFO com.example.UserService - Successfully finished processing user with ID: 123

... and so on for the other runs ...