20. Logging System

Step 1: Outline Use Cases and Constraints

Use Cases

We'll scope the problem to handle only the following use cases:

- The logging framework should support different log levels (INFO, DEBUG, ERROR, WARN).
- It should allow logging messages to multiple outputs (console, file).
- The framework should be lightweight and easily integrable with applications.
- It should support a configurable logging level to filter logs.

Out of Scope

- No support for distributed logging across multiple machines.
- No external database storage for logs.
- No UI/dashboard for log monitoring.

Constraints and Assumptions

- The system should handle concurrent log writes efficiently.
- Log file rotation is not considered in this design.
- The logging framework will support a maximum of 100 concurrent threads.

Calculate Usage

- Average log message size: ~200 bytes
- Estimated logs per second: 1000
- Expected storage usage: ~200 KB per second

Step 2: Create a High-Level Design

The logging framework consists of three primary components:

1. Logger: The main interface for applications to log messages.
2. Appender: Handles writing logs to different destinations (console, file).
3. Formatter: Formats log messages before writing.

Step 3: Design Core Components (According to the Use Cases)

Logger.java

```
import java.io.FileWriter;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;

enum LogLevel {
  INFO, DEBUG, ERROR, WARN
}

interface Appender {
  void append(String message);
}

class ConsoleAppender implements Appender {
  @Override
  public void append(String message) {
    System.out.println(message);
  }
}

class FileAppender implements Appender {
  private String filePath;

  public FileAppender(String filePath) {
    this.filePath = filePath;
  }
```

```
    @Override
    public void append(String message) {
        try (FileWriter writer = new FileWriter(filePath, true)) {
            writer.write(message + "\n");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

class Logger {
    private LogLevel level;
    private Appender appender;

    public Logger(LogLevel level, Appender appender) {
        this.level = level;
        this.appender = appender;
    }

    public void log(LogLevel logLevel, String message) {
        if (logLevel.ordinal() >= this.level.ordinal()) {
            String timestamp = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss").format(new Date());
            String formattedMessage = "[" + timestamp + "] [" + logLevel + "] " + message;
            appender.append(formattedMessage);
        }
    }
}

public class LoggingDemo {
    public static void main(String[] args) {
        Logger consoleLogger = new Logger(LogLevel.INFO, new ConsoleAppender());
        consoleLogger.log(LogLevel.INFO, "This is an info log.");
        consoleLogger.log(LogLevel.ERROR, "This is an error log.");

        Logger fileLogger = new Logger(LogLevel.DEBUG, new FileAppender("logs.txt"));
        fileLogger.log(LogLevel.DEBUG, "This is a debug log written to a file.");
    }
```

```
}
```

Step 4: Scale the Design

- **Asynchronous Logging**: Use a queue (e.g., `BlockingQueue<String>`) with a dedicated thread to write logs.
- **Log Rotation**: Implement a mechanism to rotate logs based on size or time.
- **Multiple Appenders**: Allow multiple appenders to be configured simultaneously (both console and file).
- **Performance Optimization**: Use buffering and batching to reduce I/O overhead.