10. LAB J – TEMPLATE

10.1. Purpose

We wish to implement a flexible logger class capable of logging text into various sources and be able to easily change output sources (console, file and etc...)

We will use the template design pattern to create a logger framework.

When logging information, we wish to always format the information in the same way but to postpone the decision as to where to log it, until a concrete logger is used

Lab Scenario (WHAT to do)	Define `public abstract class AbstractLogger` class with a template method and a logging method The logging method should format the text for logging and then, call the template method Define a concrete logger extending abstract logger and implementing the template method. The implementation of the template method should print the logging text to some device (console, file and etc...) Test it
Implementation Steps (HOW to do it)	Create an `AbstractLogger` class - implement a `log(String header, String body)` method - implement an abstract `log(String message)` method the logger `log(String header, String body)` method will format the text to log and pass it to the abstract template method Create a `ConsoleLogger` and `FileLogger` class - extend `AbstractLogger` - implement `log(String message)` printing the text to system.out or a file

10.2. Review of Template

Define an abstract logger class with a template method and at least one concrete logger

10.2.1. Create a new class – AbstractLogger

1. create the class AbstractLogger
2. create the logging method

```
public void log(String header, String body) {
            buffer.setLength(0);
            String message = String.format("%s: [%2$s]%n%3$s",
                    LocalDateTime.now().format(formatter), header, body);
            buffer.append(message);
            preLogHook();
            log(message);
            postLogHook();
}
```

3. create the template logging method

```
protected abstract void log(String message);
```

4. create the test code

```
        AbstractLogger logger = new ConsoleLogger();
        logger.log("alert", "memory is low");
        logger.log("info", "logger is active");
        logger.log("alert", "memory is low");
        logger.log("info", "logger is active");
        System.out.println("Done");
```

10.2.2. Create a new class – ConsoleLogger

This class will implement the template method so that logging will be directed to standard output

1. extend AbstractLogger
2. implement the template method

```
@Override
protected void log(String message) {
    System.out.println(message);
}
```

10.2.3. Optional: Create a new class – FileLogger

This class will implement the template method so that logging will be directed to a file

1. extend AbstractLogger
2. create a member variable with the file to log to

   ```
   private File logFile;
   ```

3. create the constructor

   ```
   public FileLogger(String fileName) {
      super();
      logFile = new File(fileName);
   }
   ```

4. implement the template method

   ```
   @Override
   protected void log(String message) {
       this.message = message;
       writeToFile(message, true);
   }

   private void writeToFile(String message, boolean append) {
       try (FileWriter out = new FileWriter(logFile, append);
             PrintWriter writer = new PrintWriter(out);) {
             writer.println(message);
       } catch (IOException e) {
             e.printStackTrace();
       }
   }
   ```

Note: The solution for this exercise is available in the ‘solutions’ directory.