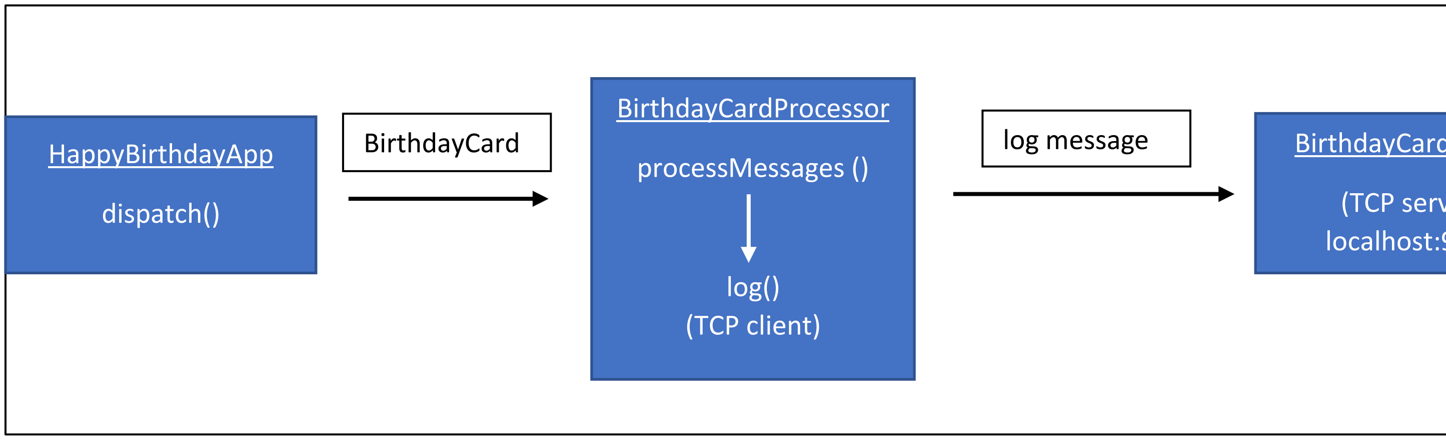
COP2805C Module 9 Practice Exercise

In this practice exercise we will use the Java Networking APIs to implement a log server which receives log messages over a TCP connection and writes them to our log file.

In the real world we would probably create this server as a standalone application (e.g. a microservice). To keep things simple, this solution uses an in-process server which runs in the HappyBirthdayApplication as a separate thread, but the implementation is basically the same as a standalone app. The basic interactions are shown in the following block diagram, the log file management has been moved out of the BirthdayCardProcessor class into a new BirthdayCardLogger class.



Here are the details of the new BirthdayCardLogger class:

```
public class BirthdayCardLogger extends Thread  {
    // allow clients to access our port
    final static int LOGPORT = 9000;

    private static final String LOGFILE = "cardlog.txt";
    private ServerSocket server;
    private Socket socket;
    private ObjectInputStream inputStream;
    private BufferedWriter cardLog;
```

The port is a constant which is publicly available for the TCP client (BirthdayCardProcessor) to access for a socket connection.

We need a ServerSocket to listen for connections, and a Socket to accept the connection. The ObjectInputStream reads our log message (a String) and we continue to use the BufferedWriter to write the message to the file.

The constructor opens the log file (create/append) and starts the server thread:

```
// constructor opens the log file and starts the server thread
public BirthdayCardLogger() {
    try {
        cardLog = Files.newBufferedWriter(Path.of(LOGFILE),
                StandardOpenOption.CREATE, StandardOpenOption.APPEND);
        start();
    } catch (IOException e) {
        System.err.println("could not open log file.");
        e.printStackTrace();
    }
}
```

The server thread listens for connections on the server socket. When a connection is received, it creates an ObjectInputStream from the connected socket. The thread then loops, waiting for a log message to arrive, which it timestamps and writes to the log file. When the client connection disconnects, the connected socket will throw an EOFException, which tells us to shut down the server thread.

```
// server thread
public void run() {

    try {
        server = new ServerSocket(LOGPORT);

        socket = server.accept();
        inputStream = new ObjectInputStream(socket.getInputStream());

        // read until client closes the connection
        while (true) {
            String logMsg = (String) inputStream.readObject();
            System.out.println("server received " + logMsg);
            LocalDateTime local =  LocalDateTime.from(
                    Instant.now().atZone(ZoneId.systemDefault()));
            String msg = local.truncatedTo(ChronoUnit.MILLIS) + logMsg;
            cardLog.write(msg);
            cardLog.newLine();
        }
    } catch (EOFException e) {
      // client closed the connection, we are shutting down
      System.out.println("logger connection closed.");
      // only flush once, so need a nested try/catch here
      try { cardLog.flush(); } catch (IOException e1) { }
```

We need to flush our BufferedWriter when exiting to make sure any pending messages are written to the file. Notice that we have to nest a try/catch in this exception handler since the flush method throws a check IOException. We don't anticipate anything going wrong here (and if it does there is not much we can do about it), so we ignore the exception in the nested handler.

The log server is aggregated by the BirthdayCardProcessor class. That class continues to implement the Logger interface as the logging client.

```
public class BirthdayCardProcessor extends Thread
        implements MessageProcessor, Logger<BirthdayCard> {

    private BirthdayCardLogger logger;
    private Socket logSocket;
    private ObjectOutputStream streamToServer;
```

The log server is started in the constructor:

```
public BirthdayCardProcessor(ConcurrentLinkedQueue<BirthdayCard> safeQueue) {
    this.safeQueue = safeQueue;

    logger = new BirthdayCardLogger();
    try {
        logSocket = new Socket("localhost", BirthdayCardLogger.LOGPORT);
        System.out.println("connected to log server");

        streamToServer = new ObjectOutputStream(logSocket.getOutputStream());
    } catch (IOException e) {
        e.printStackTrace();
    }
```

Logging still occurs in processMessages as before, but the log method is now much simpler. Timestamps and file operations are now handled by the log server thread.

```
// remove messages from the queue and process them
public void processMessages() {
    safeQueue.stream().forEach(e -> {
            // Do something with each element
            e = safeQueue.remove();
            System.out.print(e);
            log(e);
    });
}

@Override
public void log(BirthdayCard c) {
    String msg = ":greeting:" + c.getUser().getName();
    try {
        streamToServer.writeObject(msg);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

The client log socket is closed from the endProcessing method:

```
// allow external client to stop us
public void endProcessing() {
    this.stopped = true;
    try {
        logSocket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    interrupt();
```

The output for this solution has not changed from Module 8, so it is not shown here.