CSE 2104 : Object Oriented Design and Programming
Department of Computer Science and Engineering
University of Dhaka

Lab 7	**Implementing Exception Handling, File I/O, Sockets, and Threads in Projects**
Team Name	**Team Aurora**
Project Title	**MediMart**
Team Members	**1. MD.Jahidul Islam Sarker (01)** **2. Soumik Deb (11)** **3. Arin Saha Turja(37)** **4. Tukabbir Hossain Sadi(48)**

Project Brief (≈ 50 Words)

MediMart is an online medical e-commerce platform that allows customers to shop for medicines, view categories, and place orders. The platform also offers an admin login for managing inventory, processing payments, and analyzing data. With socket-based synchronization and thread management, the system ensures seamless updates and efficient multi-user handling.

Section 1 – Exception Handling

1.1 Overview

- Exception handling in MediMart ensures stability when database errors, invalid inputs, missing files, or network failures occur. The system uses a custom `AppException` to wrap low-level exceptions and provide clear messages to the user while logging technical details internally.

1.2 Implementation (Code)

Custom exception (AppException)

File: `src/utils/AppException.java`

```
public class AppException extends RuntimeException {

    public AppException(String message) { super(message); }

    public AppException(String message, Throwable cause) { super(message,
cause); }

}
```

Service layer exception handling

File: src/services/MedicineService.java

```
try {

    stmt.executeUpdate();

} catch (SQLException e) {

    FileLogger.error("DB error updating medicine: " + e.getMessage(), e);

    throw new AppException("Could not update medicine. Please try again.",
e);

}
```

UI exception handling

File: src/ui/AddMedicineForm.java, EditMedicineForm.java,
AdminDashboard.java

```
try {

    medicineService.updateMedicine(m);

} catch (AppException ex) {

    new Alert(Alert.AlertType.ERROR, ex.getMessage()).showAndWait();

}
```

1.3 Design Rationale

- **AppException** standardizes error handling across layers.
- **FileLogger** logs technical details in `logs/app.log`, not the UI.
- **UI receives friendly messages** instead of database stack traces.

Ensures robustness even on DB failure, missing file, or invalid user actions.

1.4 Challenges & Improvements

- Ensuring every database call was wrapped was time-consuming.
- Future improvement: add retry for lost DB connections or slow network.

Section 2 – File I/O Integration

2.1 Purpose

● File I/O is used for:

- Error & audit logging
- App configuration loading
- CSV exporting (admin)
- Local snapshot caching when DB fails
- Offline queue for failed DB writes

These ensure the system still functions even when SQLite or network is unavailable.

2.2 Implementation (Code)

1. File Logging

File: `src/utils/FileLogger.java`

```
private static final Path LOG_FILE = Paths.get("logs/app.log");

public static void error(String msg, Throwable t) {
    try (FileWriter fw = new FileWriter(LOG_FILE.toFile(), true)) {
        fw.write("[ERROR] " + msg + "\n");
        t.printStackTrace(new PrintWriter(fw));
    } catch (IOException ignored) {}
}
```

Where used:

- All exceptions inside `MedicineService.java`
- Socket server/client error logs
- Export operations
- Snapshot failures

Example reference:

`MedicineService.java → getAllMedicines(), addMedicine(), updateMedicine(), deleteMedicine()`

2. Configuration Loader

File: `src/utils/ConfigManager.java`

- Loads & writes `config/app.properties`
- Stores socket port, host, autosave settings
- Auto-creates config file if missing

Where used:

- **MedicineSyncServer.java (loads port)**
- **MedicineSyncClient.java (loads host/port)**

3. CSV Exporter

File: src/utils/ReportExporter.java

```
Path file = dir.resolve("medicines.csv");
BufferedWriter w = Files.newBufferedWriter(file);
```

Where used:

AdminDashboard.java – Export CSV button

2.3 Data Management Design

We ensure file consistency by using synchronized methods for reading and writing files. In case of missing or corrupted files, the system creates default files and alerts the user to fix the issue.

Section 3 – Socket Programming

3.1 Communication Model

- MediMart uses a **TCP client–server model**:
- **Admin App:** runs `MedicineSyncServer`
- **Customer App:** runs `MedicineSyncClient`

When the admin updates inventory, a `REFRESH` message is broadcast to all clients, triggering automatic medicine list refresh.

3.2 Implementation (Code)

The socket server listens for incoming client connections and broadcasts messages to refresh inventory data.

```
public class MedicineSyncServer {

    private ServerSocket server;



    public void start() {

        try {

            server = new ServerSocket(5050);

            while (true) {

                Socket client = server.accept();

                new Thread(() -> handleClient(client)).start();

            }

        } catch (IOException e) {

            FileLogger.error("Error starting sync server: " + e.getMessage(),

        e); }

    }


    private void handleClient(Socket client) {

        // handle client connection

    }


    public void broadcastRefresh() {

        for (Socket client : clients) {

            try {
                PrintWriter out = new PrintWriter(client.getOutputStream(), true);
                out.println("REFRESH");

            } catch (IOException e) {

                FileLogger.warn("Error broadcasting refresh: " + e.getMessage());

            }

        }

    }

}
```

3.3 Design Discussion

Socket programming enables:

- Real-time synchronization
- UI auto-refresh without user input
- Multi-device support (admin & customers on separate machines)

Threads ensure the server can handle multiple customers simultaneously.

Section 4 – Threads and Concurrency

4.1 Thread Use Case

Threads are used for:

- Handling each client in the socket server
- Running the socket client listener

- Background DB loading via `ExecutorService`
- Keeping JavaFX UI responsive

4.2 Implementation (Code)

Each client connection is handled in a separate thread, ensuring that the server can handle multiple clients simultaneously

.

```
public class MedicineSyncServer {

    public void start() {

        new Thread(() -> {

            while (true) {

                Socket clientSocket = serverSocket.accept();
                new Thread(() -> handleClient(clientSocket)).start();

            }

        }).start();

    }

}
```

4.3 Concurrency Design

- Uses synchronized lists, thread-safe collections for socket clients
- DB access is sequential per operation to avoid race conditions
- File writing synchronized to prevent corruption
- UI updates always executed on JavaFX Application Thread
- Socket client reconnects automatically using background loop

Feature	File	Methods Used
Custom Exception	`utils/AppException.java`	Constructor
Exception Wrapping	`services/MedicineService.java`	addMedicine(), updateMedicine(), deleteMedicine(), getAllMedicines()
Logging	`utils/FileLogger.java`	error(), warn(), info()
Config	`utils/ConfigManager.java`	get(), ensureLoaded()
CSV Export	`utils/ReportExporter.java`	exportMedicinesCsv()
Snapshot	`services/MedicineService.java`	writeSnapshot(), readSnapshot()
Offline Queue	`services/MedicineService.java`	appendFailsafe()
Socket Server	`net/MedicineSyncServer.java`	start(), broadcastRefresh()
Socket Client	`net/MedicineSyncClient.java`	start(), runLoop()
Threaded UI Loading	`ui/AdminDashboard.java`	executorService.submit()
Refresh Listener	`ui/CustomerDashboard.java`	refreshMedicines()