**Howard University**

**College of Engineering and Architecture**

**Department of Electrical Engineering & Computer Science**

**Large Scale Programming**

**Fall 2025**

**Midterm Exam**

October 20, 2025

**Instructions**

• Exam Format:  
Your examination consists of both **essay** and **programming** problems.

• Essay Questions:  
Complete all essay (rationale) questions **inline in this document**.  
My preference is **Microsoft Word (.docx)**, but **.txt** or **.pdf** are acceptable alternatives.  
Upload your completed essay file to your repository under the package:

org.howard.edu.lsp.midterm.doc

You may use **any file name**.

• Programming Problems:  
Each programming problem must be uploaded to your repository using the package specified in the question.  
For example:

org.howard.edu.lsp.midterm.question1

**• Committing Your Work:**  
If using a third-party IDE or tool to commit, commit early and often.  
Do not wait until the end of the exam to push your code.  
If you encounter problems committing, you may manually upload your code to your repository.  
If you are unable to commit or upload, you may zip your project and email it to  
bwoolfolk@whiteboardfederal.com.  
⚠️ This will result in a 20% deduction from your final exam score.

**• Citations and References:**  
You must cite all references for any material obtained from the internet.  
Any AI-generated content (e.g., ChatGPT conversations) must be included in full.  
Each package you upload must include a references document corresponding to that package’s content.  
⚠️ Failure to provide references will result in a zero for that question.

**• Exam Policy:**  
This is an OPEN BOOK, OPEN NOTES exam.  
Collaboration of any kind is strictly prohibited. Any violations will be handled in accordance with university academic integrity guidelines.

**Question 1. (20 pts.)**

**Given the following, analyze the class below and answer the below questions. This question does NOT require you to write any code.**

package org.howard.edu.lsp.studentPortalHelper;

import java.io.\*;

import java.time.LocalDate;

import java.time.format.DateTimeFormatter;

import java.util.\*;

public class StudentPortalHelper {

// Data cache (in-memory)

private static final Map<String, String> cache = new HashMap<>();

// GPA calculation

public static double computeGPA(List<Integer> grades) {

if (grades == null || grades.isEmpty()) return 0.0;

int sum = 0;

int count = 0;

for (int g : grades) { sum += g; count++; }

double avg = (double) sum / count;

// simple mapping: 90–100=A=4, 80–89=B=3, etc.

if (avg >= 90) return 4.0;

if (avg >= 80) return 3.0;

if (avg >= 70) return 2.0;

if (avg >= 60) return 1.0;

return 0.0;

}

// CSV export to disk

public static void exportRosterToCsv(String filename, List<String> names) {

try (PrintWriter pw = new PrintWriter(new FileWriter(filename))) {

pw.println("name");

for (String n : names) {

pw.println(n);

}

} catch (IOException e) {

System.err.println("Failed to export roster: " + e.getMessage());

}

}

// Email formatting

public static String makeWelcomeEmail(String studentName) {

return "Welcome " + studentName + "! Please visit the portal to update your profile.";

}

// Date formatting (UI concern)

public static String formatDateForUi(LocalDate date) {

return date.format(DateTimeFormatter.ofPattern("MM/dd/yyyy"));

}

// Payment processing (stub)

public static boolean processTuitionPayment(String studentId, double amount) {

if (amount <= 0) return false;

// pretend to call external gateway...

return true;

}

// Password strength check (security)

public static boolean isStrongPassword(String pwd) {

if (pwd == null || pwd.length() < 8) return false;

boolean hasDigit = false, hasUpper = false;

for (char c : pwd.toCharArray()) {

if (Character.isDigit(c)) hasDigit = true;

if (Character.isUpperCase(c)) hasUpper = true;

}

return hasDigit && hasUpper;

}

// Ad-hoc caching

public static void putCache(String key, String value) {

cache.put(key, value);

}

public static String getCache(String key) {

return cache.get(key);

}

}

**Tasks:**

Using one or more Arthur Riel heuristics, analyze whether the StudentPortalHelper class demonstrates high or low cohesion.  
a) Should a well-designed class have high or low cohesion? Explain and defend your answer. (5 pts.)

A well-designed class should have high cohesion. High cohesion means the class’s methods and fields are closely related and focused on a single responsibility or cohesive set of responsibilities. High cohesion makes code easier to understand, easier to test, and easier to reuse.

b) Based on your analysis, discuss—**only if you believe changes are needed**—how you would reorganize or redesign the class to improve its structure. Your answer should (1) identify the class as having high, low or perfect cohesion and (2) describe a **general approach** to refactoring the class. If you believe the class already has good cohesion, justify why no changes are necessary. (15 pts)

*(If you believe the class already has good cohesion, justify why no changes are necessary.)*

I identify the class as having low cohesion

GPAService (business logic): computeGPA(List<Integer>). Unit-testable logic.

RosterCsvExporter (I/O): exportRosterToCsv(String filename, List<String> names). Handles file I/O and formatting.

EmailFormatter (presentation): makeWelcomeEmail(String studentName) and other email templates.

UiDateFormatter or simply use a DateFormatter utility for UI-specific formatting: formatDateForUi(LocalDate).

PaymentProcessor or TuitionGateway (integration/adapters): processTuitionPayment(String studentId, double amount) — this should be an interface with implementations that call external gateways; production implementation would handle retries, logging, exceptions.

SecurityUtils (password policy): isStrongPassword(String pwd).

CacheManager (caching): a non-static cache implementation behind an interface (e.g., Cache<K,V>), possibly injectable, with thread-safety and eviction policy if needed.

Refactor Details

• Replace static methods with instance classes where it makes sense (e.g., PaymentProcessor should be an instance or injected dependency to allow mocking/test stubbing). Utility methods (pure functions) can remain static in dedicated utility classes (e.g., SecurityUtils.isStrongPassword), but avoid lumping unrelated utilities in a single god-class.

• Remove the private static final Map<String,String> cache from this helper and put caching behind a CacheManager interface. Make cache thread-safe if used across threads (e.g., ConcurrentHashMap) and add lifecycle/eviction if needed.

• Make I/O code (CSV export) fail loudly (throw checked/unchecked exceptions) or accept an OutputStream/Writer for greater testability; avoid printing to System.err in library code.

• Move UI concerns (date formatting and welcome email wording) into presentation layer code (or a UiFormatter) and keep business logic independent of UI formatting.

• Add unit tests per class. Use dependency injection to supply test doubles for PaymentProcessor and CacheManager.

Benefits

• Each class has higher cohesion and one reason to change.

• Easier unit testing and mocking.

• Improved readability and maintainability.

• Enables reuse of individual components (e.g., SecurityUtils) without importing unrelated code.

**Question 2. (20 pts.)**

Write a class AreaCalculator in the package org.howard.edu.lsp.midterm.question2 with the following **overloaded methods**: This should be uploaded to your repo.

// Circle area

public static double area(double radius)

// Rectangle area

public static double area(double width, double height)

// Triangle (base & height) area

public static double area(int base, int height)

// Square (side length) area

public static double area(int side)

**Requirements:**

Each method should compute and return the correct area.

• Circle area: π (use class Math.PI) × r²

• Rectangle area: width × height

• Triangle area: ½ × base × height

• Square area: side²

• For all methods: throw an IllegalArgumentException if any dimension is ≤ 0.

Create a class named Main that invokes each overloaded method **statically** to produce **exactly** the following output:

Circle radius 3.0 → area = 28.274333882308138

Rectangle 5.0 x 2.0 → area = 10.0

Triangle base 10, height 6 → area = 30.0

Square side 4 → area = 16.0

Finally, invoke **at least one** of the area methods with a value that causes an IllegalArgumentException to be thrown.

• Catch the exception using a try/catch block.

• Print an **error message** to System.out. (Any message is fine.)

Briefly (2–3 sentences as a comment in class Main) explain if **overloading** or simply use methods with different names, i.e., rectactangleArea, circleArea, etc..

**Category**

**Description**

**Points**

**1. Implementation**

Correct use of **method overloading** (same name, different signatures), correct formulas, and proper exception handling in each method.

**10**

**2. Program Behavior**

Main correctly invokes all methods statically, produces the required output exactly, and includes a working exception demonstration.

**6**

**3. Conceptual Understanding**

Brief explanation of why or why not overloading is the better design choice.

**4**

**Question 3.**

**Given the following, answer the below questions.**

**(20 pts.)**

**Given:**

A car manufacturer uses Java software to track current vehicles being built. The UML diagram below shows an excerpt of the current software structure. You should assume the presence of other appropriate fields and methods

**Car**

getTrimLevel()

**Engine**

accelerate()

getFuelLevel()

**Base**

**Sports**

**Luxury**

**Electric**

**Petrol**

**Car**

getTrimLevel()

**Engine**

accelerate()

getFuelLevel()

**Base**

**Sports**

**Luxury**

**Electric**

**Petrol**

Each car can be built to one of three trim levels: Base, Luxury or Sport. They can also be configured with an electric or petrol engine. At various points in the manufacturing process the customer can choose to change the trim level.

**Task:**

• Explain in detail why the current structure does or does not support this. (10 pts.)

• Describe how to refactor the structure to allow trim-level change for a car to dynamically change. Hint: How would you modify Car to use composition to solve the problem? (10 pts.)

From the UML diagram, the trim levels (Base, Sports, Luxury) appear to be separate subclasses (or types) tied to Car in an inheritance-like arrangement. If trim levels are implemented by subclassing Car (e.g., BaseCar, SportCar, LuxuryCar), then changing a car’s trim at runtime is **not supported** in-place. You must create a new object of the appropriate subclass and replace the old one. That approach couples behavior/state with type identity and prevents a simple dynamic change to trim on the same Car instance.

This is a problem because manufacturing requires that an existing vehicle instance be reconfigured during production (customer changes trim). Subclassing for trim level mixes a runtime configuration detail with a static type definition, making dynamic reconfiguration cumbersome and error-prone while violating the Open/Closed and Single Responsibility principles that support flexible design.

**Question 4. (30 pts.)**

Design and implement a small **smart-campus device** system showing both **class inheritance** (concrete classes extend a common abstract class) and **interface implementation**.

**1) Abstract Base Class — Device**

**The following Device class is partially implemented for you.  
You must not modify this code, but you will use it in your subclasses:**

package org.howard.edu.lsp.midterm.question4;

public abstract class Device {

private String id;

private String location;

private long lastHeartbeatEpochSeconds;

private boolean connected;

// PROVIDED CONSTRUCTOR

public Device(String id, String location) {

if (id == null || id.isEmpty() || location == null || location.isEmpty()) {

throw new IllegalArgumentException("Invalid id or location");

}

this.id = id;

this.location = location;

this.lastHeartbeatEpochSeconds = 0;

this.connected = false;

}

public String getId() {

return id;

}

public String getLocation() {

return location;

}

public long getLastHeartbeatEpochSeconds() {

return lastHeartbeatEpochSeconds;

}

public boolean isConnected() {

return connected;

}

protected void setConnected(boolean connected) {

this.connected = connected;

}

public void heartbeat() {

this.lastHeartbeatEpochSeconds = System.currentTimeMillis() / 1000;

}

public abstract String getStatus();

}

**You will extend this class** in your DoorLock, Thermostat, and Camera implementations.  
All subclasses must call super(id, location) in their constructors.

**2) Capability Interfaces (behaviors only)**

**Networked**

void connect();

void disconnect();

boolean isConnected();

Behavior:

• connect() brings the device online by setting connected = true.

• disconnect() sets connected = false.

• isConnected() reports the current connection state.  
(Concrete classes may satisfy this using Device’s protected setter and public getter.)

**BatteryPowered**

int getBatteryPercent(); // 0..100

void setBatteryPercent(int percent);

Behavior:

• getBatteryPercent() returns current battery %.

• setBatteryPercent(int) updates it; throw IllegalArgumentException if outside 0..100 inclusive.

**3) Concrete Devices (must extend Device and implement interfaces)**

**All fields must be private. Implement methods exactly as specified.**

**• DoorLock — extends Device, implements Networked, BatteryPowered**

**Private fields**

private int batteryPercent;

**Constructor**

public DoorLock(String id, String location, int initialBattery)

• Call super(id, location).

• Initialize battery by calling setBatteryPercent(initialBattery) (enforces 0..100).

**Implemented methods**

// Networked

@Override public void connect() { setConnected(true); }

@Override public void disconnect() { setConnected(false); }

@Override public boolean isConnected() { return super.isConnected(); }

// BatteryPowered

@Override public int getBatteryPercent() { return batteryPercent; }

@Override public void setBatteryPercent(int percent) {

if (percent < 0 || percent > 100) throw new IllegalArgumentException("battery 0..100");

this.batteryPercent = percent;

}

// Status

@Override public String getStatus() {

String connStatus = isConnected() ? "up" : "down";

return "DoorLock[id=" + getId() + ", loc=" + getLocation() +

", conn=" + connStatus + ", batt=" + batteryPercent + "%]";

}

**B) Thermostat — extends Device, implements Networked**

**Private fields**

private double temperatureC;

**Constructor**

public Thermostat(String id, String location, double initialTempC)

• Call super(id, location).

• Initialize temperatureC to initialTempC.

**Accessors**

public double getTemperatureC();

public void setTemperatureC(double temperatureC);

**Implemented methods**

// Networked

@Override public void connect() { setConnected(true); }

@Override public void disconnect() { setConnected(false); }

@Override public boolean isConnected() { return super.isConnected(); }

// Status

@Override public String getStatus() {

String connStatus = isConnected() ? "up" : "down";

return "Thermostat[id=" + getId() + ", loc=" + getLocation() +

", conn=" + connStatus + ", tempC=" + temperatureC + "]";

}

**C) Camera — extends Device, implements Networked, BatteryPowered**

**Private fields**

**private int batteryPercent;**

**Constructor**

public Camera(String id, String location, int initialBattery)

• Call super(id, location).

• Initialize battery by calling setBatteryPercent(initialBattery).

**Implemented methods**

// Networked

@Override public void connect() { setConnected(true); }

@Override public void disconnect() { setConnected(false); }

@Override public boolean isConnected() { return super.isConnected(); }

// BatteryPowered

@Override public int getBatteryPercent() { return batteryPercent; }

@Override public void setBatteryPercent(int percent) {

if (percent < 0 || percent > 100) throw new IllegalArgumentException("battery 0..100");

this.batteryPercent = percent;

}

// Status

@Override public String getStatus() {

String connStatus = isConnected() ? "up" : "down";

return "Camera[id=" + getId() + ", loc=" + getLocation() +

", conn=" + connStatus + ", batt=" + batteryPercent + "%]";

}

**4) Provided Driver**

**Do not modify this file. Your classes must compile and run with it unchanged.**

package org.howard.edu.lsp.midterm.question4;

import java.util.\*;

public class Main {

public static void main(String[] args) {

Device lock = new DoorLock("DL-101", "DormA-1F", 85);

Device thermo = new Thermostat("TH-202", "Library-2F", 21.5);

Device cam = new Camera("CA-303", "Quad-North", 72);

// === Invalid battery test ===

System.out.println("\n== Exception test ==");

try {

Device badCam = new Camera("CA-404", "Test-Lab", -5);

System.out.println("ERROR: Exception was not thrown for invalid battery!");

} catch (IllegalArgumentException e) {

System.out.println("Caught expected exception: " + e.getMessage());

}

// === Heartbeat demonstration ===

System.out.println("\n== Heartbeat timestamps BEFORE ==");

for (Device d : Arrays.asList(lock, thermo, cam)) {

System.out.println(d.getId() + " lastHeartbeat=" + d.getLastHeartbeatEpochSeconds());

}

lock.heartbeat();

thermo.heartbeat();

cam.heartbeat();

System.out.println("\n== Heartbeat timestamps AFTER ==");

for (Device d : Arrays.asList(lock, thermo, cam)) {

System.out.println(d.getId() + " lastHeartbeat=" + d.getLastHeartbeatEpochSeconds());

}

// === Base-class polymorphism ===

List<Device> devices = Arrays.asList(lock, thermo, cam);

System.out.println("\n== Initial status via Device ==");

for (Device d : devices) {

System.out.println(d.getStatus());

}

// === Interface polymorphism: Networked ===

System.out.println("\n== Connect all Networked ==");

for (Device d : devices) {

if (d instanceof Networked) {

((Networked) d).connect();

}

}

// === Interface polymorphism: BatteryPowered ===

System.out.println("\n== Battery report (BatteryPowered) ==");

for (Device d : devices) {

if (d instanceof BatteryPowered) {

BatteryPowered bp = (BatteryPowered) d;

System.out.println(d.getClass().getSimpleName() + " battery = " + bp.getBatteryPercent() + "%");

}

}

// === Final status check ===

System.out.println("\n== Updated status via Device ==");

for (Device d : devices) {

System.out.println(d.getStatus());

}

}

}

**5) Brief Rationale (2–4 sentences)**

• Why is Device defined as an abstract class?

Device provides shared state (id, location, heartbeat time, connected flag) and some common behavior (constructor validation, heartbeat(), getLastHeartbeatEpochSeconds(), protected setConnected(boolean)) while leaving device-specific status reporting to subclasses via the abstract getStatus() method. Making it abstract enforces a common contract and prevents instantiation of a generic device.

• How do the Networked and BatteryPowered interfaces add behavior to your concrete classes?

The interfaces define behavior contracts (connect/disconnect/isConnected and getBatteryPercent/setBatteryPercent) which concrete classes implement according to their own state; this allows polymorphism by interface so callers can treat devices generically (e.g., operate on all Networked devices or report BatteryPowered devices).

• Is this design an example of *multiple inheritance* in Java? Explain why or why not.

No, Java does not allow multiple class inheritance. This design uses single inheritance for shared Device functionality and multiple *interface* implementation to achieve multiple-behavior composition. The classes inherit from one base class and implement multiple interfaces, which is allowed and is not multiple inheritance of implementation in Java.

**Grading (30 pts)**

**Category**

**Description**

**Points**

**Implementation**

Correct use of inheritance and interfaces; meets all required method signatures and behaviors; uses the provided Device constructor; correctly implements Networked and BatteryPowered; uses setConnected(boolean) properly; validates inputs.

**15**

**Program Behavior**

Code compiles and runs with the provided Main.java unchanged; heartbeat behavior works; base-class and interface polymorphism demonstrated; exception thrown for invalid battery input; getStatus() output matches required formats.

**9**

**Rationale**

Clear, thoughtful, and specific answers to the four questions above. References to the student's own code are present. Shows conceptual understanding of abstraction, interface-based behavior, and multiple inheritance in Java.

**6**

**Question 5 (10 pts)**  
**Reflection on AI Use in Learning and Problem Solving**

Discuss your personal experience using **AI tools** (such as ChatGPT, GitHub Copilot, or others) before and during this course.  
In your response, address the following points:

• How have you used AI to support your learning or programming in this course?

• What benefits or limitations did you encounter?

• Looking ahead, how do you expect AI to influence the way you solve problems **academically or professionally**?

Your answer should be **1–2 well-developed paragraphs.**

Before this course I used AI tools like ChatGPT and code-completion assistants to prototype ideas, get quick explanations of algorithms, and find concise examples of syntax I was unfamiliar with. During the course I used AI to help clarify assignment instructions, generate small code templates, and to practice answering technical questions; it sped up the exploratory stage but I always validated outputs and adapted them to class constraints.

The main benefits I found were faster iteration and succinct summaries; the limitations were occasional factual mistakes, missing context-specific constraints, and the need to verify correctness and style.

Looking ahead, I expect AI to become a key partner in how I approach both academic and professional problem-solving. Rather than replacing critical thinking, it will serve as a catalyst for it by helping me quickly analyze data, generate alternative solutions, and identify blind spots I might overlook on my own. In the classroom, AI can streamline research and coding workflows, allowing me to focus more on conceptual understanding. Professionally, it will enhance collaboration and efficiency by automating repetitive tasks, improving decision-making, and providing data-driven insights that support more creative and informed problem-solving.