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 always demonstrate high cohesion, meaning that all its methods and data members work together to achieve a single, well-defined purpose. High cohesion leads to code that is easier to maintain, test, and understand. When a class performs too many unrelated tasks, it becomes harder to modify or reuse, violating Arthur Riel's heuristic that "a class should capture one and only one key abstraction."

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.)

The StudentPortalHelper class shows low cohesion because its methods serve many unrelated purposes like calculating GPAs, exporting CSV files, formatting emails, processing payments, checking passwords, and handling caching. These functions span multiple domains: academics, I/O operations, UI formatting, financial transactions, and security. Since none of these responsibilities share common data or behavior, the class violates Riel's heuristic that "all services of a class should be closely related to that abstraction." To improve cohesion, the class should be refactored into smaller, domain-focused classes. For instance, GPA-related logic could belong in a GpaCalculator class, CSV export in a RosterExporter, payments in a PaymentProcessor, password checks in a PasswordPolicy class, and date formatting in a UI helper. The ad-hoc caching system should either be its own CacheService or removed in favor of a proper caching mechanism. This restructuring would align each class with a single, consistent responsibility and follow Riel's guideline that "classes that share common data should be grouped together, and unrelated features should be separated."

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) × r^2
- Rectangle area: width × height
- Triangle area: ½ × base × height
- Square area: $side^2$
- 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.

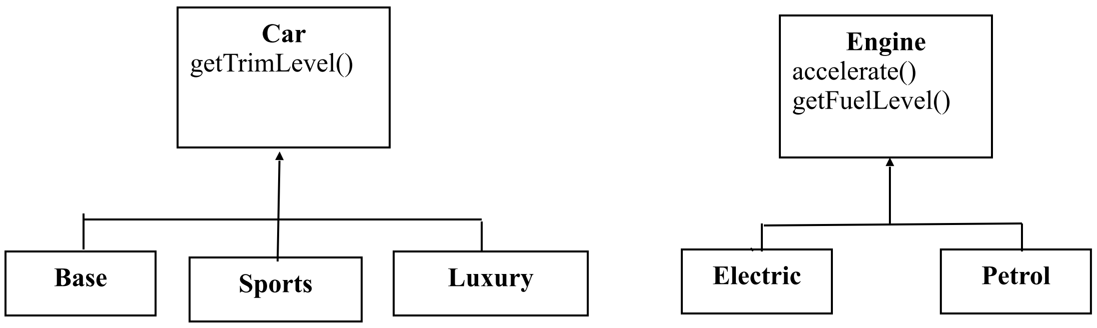
- Catch the exception using a try/catch block.
- Print an **error message** to System.out. (Any message is fine.)

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



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:
a) Explain in detail why the current structure does or does not support this. (10 pts.)
The current design uses inheritance to represent trim levels (Base, Sports, Luxury) as subclasses of Car. This structure assumes that a car's trim level is a permanent property fixed at object creation. Because inheritance defines an "is-a" relationship, a Sports car is a Car, not a Car that can become a Luxury model later. Therefore, the design tightly couples behavior to subclass type, preventing dynamic modification during runtime. If new trim levels are introduced, new subclasses must be created and tested separately. The design also increases code duplication, since common car behavior must exist across all subclasses, reducing flexibility and maintainability.

b) 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.)
To support changing trims dynamically, the Car class should use composition rather than inheritance. Instead of making each trim level a subclass, introduce a new class hierarchy such as TrimLevel with subclasses BaseTrim, LuxuryTrim, and

SportTrim. Then give the Car class a field like **private TrimLevel trim;** The Car class can delegate behavior to the TrimLevel object and expose a setter method to switch trims at runtime: **public void setTrimLevel(TrimLevel newTrim) {**
   this.trim = newTrim;
}
This design makes trim behavior interchangeable, which allows algorithms or behaviors to change independently of the object using them. As a result, the car can easily switch from Base to Luxury without creating new subclasses, improving flexibility, scalability, and code reuse.

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.

A) 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?
- How do the `Networked` and `BatteryPowered` interfaces add behavior to your concrete classes?
- Is this design an example of *multiple inheritance* in Java? Explain why or why not.

The Device class is defined as an abstract class because it represents a general blueprint for all smart-campus devices, providing shared fields (like id, location, and connection state) and common methods such as heartbeat() while leaving device-specific behavior (getStatus()) to be implemented by subclasses. The Networked and BatteryPowered interfaces add specific behavioral capabilities that different devices may or may not have, such as network connectivity or battery management, without forcing unrelated classes to inherit unnecessary code. This design demonstrates behavioral composition rather than true multiple inheritance, since Java allows a class to extend only one superclass but implement multiple interfaces to combine different functionalities cleanly.

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:

1. How have you used AI to support your learning or programming in this course?
2. What benefits or limitations did you encounter?
3. 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 occasionally used AI tools like ChatGPT to clarify programming concepts or debug small coding errors, but I didn't fully understand how to use them effectively for learning. During this course, I began using AI more intentionally, to review code logic, understand object-oriented design, and explore how different

programming structures work in Java. ChatGPT, for instance, helped me break down complex assignments and understand UML diagrams, while GitHub Copilot provided helpful code suggestions and syntax corrections. These tools made it easier to experiment, learn faster, and reinforce what I learned in lectures and labs.
However, I have noticed some limitations. AI tools can sometimes generate incomplete or overly general explanations that require verification, so I learned to double-check outputs against course materials or my own reasoning. Despite that, the benefits far outweighed the drawbacks. Looking ahead, I expect AI to play a major role in how I solve problems, both academically and professionally, by serving as a collaborative assistant for brainstorming, debugging, and optimizing solutions. Rather than replacing learning, AI will continue to help me think critically and approach programming challenges more efficiently.