**Speech draft:**

Hello everyone, my name is Ben Pare. Today, I will be reviewing some code projects I worked on in previous classes.

First, I want to briefly mention the classes involved:

* **CS 465** focused on full-stack development. I earned a good grade on the final project, though I did struggle with some parts along the way.
* **CS 330** was a recent project dealing with 3D-like images. I’m not sure if I was able to make it fully work—there might have been some dependency issues.
* **CS 320** involved writing code in Java with testing frameworks like JUnit and Maven.

Let’s start with **CS 320**. You can open these projects using various programs like Visual Studio or Eclipse. The exact IDE doesn’t matter much for our review today—we are focusing on the **code logic and structure**, not whether the program runs perfectly on your machine.

Looking at CS 320, I noticed there are multiple folders with the same name, like AppointmentServiceBenPare(2) and one without the "(2)". I believe I accidentally created duplicate folders, so tidying these up would help with organization.

There are several important files:

* Appointment.java, AppointmentService.java, AppointmentServiceTest.java, AppointmentTest.java,
* Contact.java, ContactService.java, ContactServiceTest.java, and others.

The Appointment.java class is a good starting point, as it likely serves as the foundation for the other files. In this class, the code performs several checks to validate appointment data and stores that data in variables. It uses getters and setters, which makes data access straightforward and efficient. Since there are no loops here, the runtime is roughly O(N), which is good for performance.

The AppointmentService.java acts more like a front-end interface, using a HashMap to store appointments. The if-statements check conditions, and the code returns early to avoid unnecessary else blocks. This structure makes the code modular and easier to read.

For testing, JUnit tests validate whether the code works as expected. The tests include setup steps, where appointments are manually created to check the program’s functionality.

**Brief explanation of CS-330:**

This project focuses on modeling simple 3D objects and creating a basic 3D scene.

Starting with the **Camera class**, it defines the parameters of a camera, such as its position and movement capabilities—like moving up, down, left, right—as well as controlling pitch, speed, and zoom. These are the basics needed to make the camera move around in 3D space. The class also includes a destructor to free up resources when the camera object is destroyed, which is important to prevent memory issues.

The camera class handles input processing from the mouse and keyboard, allowing dynamic control of the camera. Functions like updateCameraVectors() and setOrientation() are critical because they update where the camera is pointing and how it moves, which is essential for navigating the 3D scene.

Next, in the **mainCode.cpp** file, the program starts by testing whether the initial loading is successful. If any step fails, the program exits with a failure status. One thing I liked here is that the comments are well written and explain the code clearly.

The program initializes two important libraries: **GLFW** for managing windows and input, and **GLEW** for loading OpenGL extensions. These are standard in OpenGL projects.

We then create a **Scene Manager**, responsible for managing textures and 3D shapes. The Scene Manager has a constructor to set things up and a destructor to clean up resources. One key task is loading textures, like a wood texture for an object in the scene. If loading the texture fails, the program exits to avoid errors later on.

The project also includes a **Shader Manager**. Shaders are programs that run on the GPU to control how objects are rendered—they add effects like lighting and shadows, which make the scene look more realistic. The Shader Manager loads and compiles these shaders, and checks for errors to ensure everything is working correctly. There are lots of error checks in the code, probably because I encountered linker errors during development and added these checks to handle those problems.

In general, the code is well organized and modular: functions are separated by responsibility, which makes testing and debugging easier. For example, the **Texture.cpp** file handles creating and setting textures, positioning them correctly, and checking for failures. It uses getter methods like getId() to retrieve texture identifiers and includes a destructor to properly delete textures when they’re no longer needed.

An important file is the pom.xml, which manages dependencies for Maven. I have two pom.xml files, possibly one for different parts of the project or tests. It’s good practice to keep these updated to ensure the project builds correctly.

Overall, the code is fairly organized but could benefit from better comments to explain some logic more clearly. I also noticed that the variables have clear, meaningful names, and I don’t see many redundant or unused variables.

Regarding memory management, the use of getters and setters helps keep data consistent and avoid memory leaks. However, I’m unsure if there’s a timeout feature implemented.

here, and adding one might improve reliability, especially for external resource access.

**Code Review Speech CS 465**

*Structure*  
Overall, the code implements the core design objectives and mostly follows common coding standards. The structure is generally clear and modular, but there are a few inconsistencies — for example, some routes and routers are declared multiple times or imported from the same file, which could be streamlined further.  
I noticed some potential missing pieces or incomplete wiring of routes, such as multiple routers referencing the same index file, which may cause confusion or unexpected behavior. This is an area I plan to revisit and clarify in future iterations.  
While most repeated code was minimized, there are still some areas where DRY (Don’t Repeat Yourself) principles could be improved. I’ve removed leftover debugging stubs, but I suspect some unreachable code may still exist and needs pruning.

*Documentation*  
The code is commented adequately, but there are places where comments could be more detailed to explain the reasoning behind certain decisions. Some comments might be outdated or inconsistent with recent code changes, so that’s an area for ongoing maintenance.

*Variables*  
Variable naming is mostly clear and consistent, but I discovered some variables that could be more descriptive or consolidated. There are a few unused imports or variables that slipped through, likely due to refactoring and code evolution.

*Arithmetic Operations*  
The project doesn’t heavily involve complex arithmetic, so this wasn’t a major focus. However, I’m aware that input validation around numerical or string data could be tighter to avoid unexpected edge cases.

*Loops and Branches*  
Most loops and conditional branches are complete and properly nested. However, I found a few conditionals where default or fallback cases were missing, which could lead to undefined behavior if unexpected inputs occur. I intend to review these areas and add the necessary safeguards.

*Defensive Programming*  
Input validation is in place but could be more rigorous, especially for routes receiving external data. I recognize that bounds checking and error handling need to be more consistent throughout the application. Some asynchronous error handling is minimal and would benefit from more robust try-catch structures or middleware error handling.  
Resource cleanup and state management are handled, but I would like to improve this further to ensure no resource leaks or stale connections remain under error conditions.