- Hello, my name is Harrison Bergeron and I am going to be reviewing three separate code artifacts to enhance for the CS-499 capstone project.

- For my first artifact, I am using a Java Swing application from CS 250 demonstrating an interactive GUI with button widgets and images to control a slide show interface depicting popular wellness travel destinations. You can see the functionality visually when I run the program [Run the program]. A window opens that I can move around, switch between slides using next and previous buttons, and close the program using the “X” button.

**Structure**

- Code quality aside and given the requirements for a slideshow application with images, text descriptions, and interactive slide controls, the program achieves the desired functionality in this artifact.

- As seen in the private fields at the top of the class as well as for variables in method scopes, variables are named using lowerCamelCase according to the Google Java Style Guide. Method names also conform to this style.

- Whitespace is placed between operators and symbols as necessary to improve readability. Tab indentation is similarly utilized to improve readability and separate class, method, loop, and conditional scopes.

- The code also follows the Java standard to make sure that every class gets its own source file except for nested classes.

(125 & 145) The code is well structured and consistent in style with naming conventions, whitespace, and formatting, although some of the conditional checks within these if-else branches omit whitespaces between the equality operator and its arguments as well as between braces.

- There are no unneeded procedures or unreachable code within this source file. All unnecessary testing code or method stubs were removed upon completion of this artifact.

(81 & 163) Library functions and classes are already used efficiently to reduce excess code and issues of “reinventing the wheel.” Swing action listeners are used to handle button logic automatically. The slide show application is launched in a thread-safe manner using EventQueue.invokeLater() instead of attempting to synchronize code manually.

(122 & 142) The methods getResizeIcon() and getTextDescription() contain repetitive code for setting an image style through HTML. The lines that set the “image” and “text” strings to their corresponding HTML styles can be condensed into individual functions that automatically format this repetitive string.

- Storage use is efficient in this application because there are no private member fields that are unused and taking up memory unnecessarily. All resources are utilized upon launching the application.

- Several string and numerical constants are used in the application in place of symbolics. Some are justified and some are not.

(44 & 56) The textPane object is initialized with a constant background color and the title of the window is initialized to a constant string. Constants like these are acceptable, although they might be better off being initialized as their own fields within the class in case they need to be reused.

(45 & 54 & 122) The size of the text pane and window are initialized with constants, but they should probably be computed on the fly to allow for more dynamic sizing functionality of the interface. The getResizeIcon() method includes highly repetitive string constants that can be stored in a single method or variable, as well as width and height styles that should probably be computed on the fly for UI flexibility.

(66) This for loop contains an arbitrary last index of 5 to create the 5 slides present in the program. This constant shows up in other parts of the program, which makes it difficult to add more slides if necessary. Storing this value in one place or adding methods to facilitate adding new slides easily would be a better design.

- Because of the simplicity of this application, the classes and routines are concise enough to not be split up anymore.

**Documentation**

(14) Documentation for method signatures is detailed and sufficient for this application, although there is no class documentation for SlideShow. Additionally, GUI widget variables are lacking enough of a description to make it clear what purpose they serve.

**Variables**

(125 & 142 & 167) Most variables have meaningful and clear names, but the methods getResizeIcon() and getTextDescription() have parameters named “i” that are somewhat vague and need to be more descriptive. The “ss” variable at the bottom of the source code could be named to something like “appInstance” to convey that it is the main running instance of the program.

- All variables have appropriate types and none are inappropriately cast to other types.

(62) The only seemingly redundant variables are the card layouts. In actuality, these two instances have to be separate for the two widgets to function correctly despite the fact that the layouts are the same.

(66) One arguable issue with the lblSlide and lblTextArea fields is that they don’t refer to specific widgets, but are simply used as placeholder variables within this loop. It might be better to remove these widgets from the private class fields and only initialize them within the local scope of this loop. These variables are also unnecessarily initialized at the beginning of the initComponent() method even though they will be reinitialized at the beginning of this loop.

**Arithmetic Operations**

- The code completely refrains from comparing floating point numbers using the equality operator.

(44) The only floating point constants appear here, where rounding errors will not result in any noticeable difference in the background color.

- There are no operations involving floating point addition, subtraction, multiplication, or division that could result in unstable numerical arithmetic.

**Loops and Branches**

(66 & 125) The for loop that adds slides to the UI is correct and loops for the expected number of total slides. The if-else chains here are accomplished in such a way that excess indentation is not required for each else branch to improve readability.

(125 & 145) The if-else chains here are correct and clear, although they could more appropriately be replaced with a switch statement to test for each index.

(125 & 145) Not all cases are considered for these if-else chains, including values beyond 5 or less than 1. This could be resolved by including an else branch at the end of the chain to handle erroneous inputs.

(66) The loop to add slides to the UI iterates from i = 1 to 5 inclusive, and there are no exceptions. It will run for exactly 5 iterations and no infinite loops could occur. i is not altered within or after the loop.

(66) All of the statements inside the loop are necessary within this scope and cannot be placed outside.

**Defensive Programming**

(122 & 142) No indices need to be checked before array or record bounds except for at the getResizeIcon and getTextDescription methods. As I stated previously, it would be a good idea to add an else branch to check for out of bounds values of i. Additionally, the image resources are not checked to see if the files exist before being inserted into the image styling string.

(162) There is no input data for the program, although there is a string array of arguments as the parameter to the main method. Since these arguments are not utilized at runtime, they are not validated. If anything, the program could produce a warning message for ignored arguments that the user has supplied to the program.

- This program has no outputs aside from the graphical user interface that is displayed for the user. The buttons are assigned listeners to respond to user input instantly and all widgets are initialized to display content to the user upon launching the program.

- The correct data is operated on within each statement in this code, and no external data is accessed except for the images loaded in for each slide.

- Every memory allocation is deallocated properly when variable references are no longer used because Java has a garbage collector. There are some instances where libraries will have methods to manually dispose resources, but this application uses no such libraries.

(30) A HeadlessException is thrown at the constructor of the application in case the program is executed within an environment where a keyboard, mouse, or display is not present.

- As I stated previously, the images depicted in each slide should be checked for existence before being included in the UI.

- All resources will be closed and disposed of automatically when this application exits, and no files or devices are modified during the execution of this program whose state would need to be saved before termination.

**Enhancements and Course Outcomes**

- In addition to addressing the aforementioned code weaknesses, limitations, and vulnerabilities, I will add entirely new functionality in the process of enhancing this artifact. This application was written using the Java programming language and its built-in GUI library Swing. In order to meet the outcomes CS-499-04 and CS-499-05, I will port this application to C++ and use a different GUI API called Qt. The new application will mimic the functionality and layout of the Java Swing application. I will need to manually transpile this Java code to C++ while integrating the proper programming practices for C++ and using appropriate Qt equivalents to Swing components and widgets. I will meet the CS-499-04 outcome by iteratively developing and testing the new codebase for this application in the same pattern I used while designing the original artifact using principles of the software development life cycle. I will use the Qt API to create industry-standard software designs that adhere to [common design patterns](https://sourcemaking.com/design_patterns) (*Design Patterns and Refactoring*) that can be found through online sources. I will improve upon existing Java code when designing the C++ implementation of this application to be more robust and achieve a common industry goal of porting functionality to another language that is more efficient. I will also target the CS-499-05 outcome by mitigating design flaws and security vulnerabilities within the original artifact. Better flexibility and room for potential future code changes will be accounted for in the enhanced version of this artifact through the implementation of a more scalable and modular design.

**References**

SourceMaking. (n.d.). *Design patterns and refactoring*. SourceMaking. Retrieved July 5, 2022, from https://sourcemaking.com/design\_patterns