**Intro**

I designed this project while balancing between good practices and good program design and simplicity. When writing code, I try to practice the "clean code" principles. Therefore, on some occasions, I preferred to extend specific actions, even if I could write them more shortly.

Also, some of this app design is a bit of "overkill" for this kind of small project, but I wanted to show better design practices for larger-scale projects.

**Readability and Maintainability** – Sometimes I prefer to extend the code a bit just to make it easier for other fellow developers to follow read through my code more easily and make it easier to understand the code just by reading it (self-documentary code) (i.e I took out computational operations into variables to clear the purpose of the calculation. I write the getPlayerAudio(player) function for HTML structure changes flexibility).

**Reusability** – all the functions were designed to be used in more than one place, without being dependent on the context in which they are being called. (i.e I implemented two separate functions for add and remove className ("off", "active") in the internal functions instead of just toggle it.)

**Abstract/Concrete separation** – I have tried in most cases to separate the logic from the concrete operations to follow the logic. This allows changing the underlying implementation without having to impose risks to the functionality of the app (by touching the logic code)

**Project Overview**

I designed this project with 2 main entities in mind: Player and Audio. Player is a container (very much like an audio player in your computer) it holds functionality to play audio. Audio is the actual sounds that can be played inside the player.

A user will interact only with the player, and the player will control the audio in turn.

Each pad in the UI is a player that holds one audio file. On each pad click, the player's audio-playing functionality is being toggled on and off (function togglePlayer).

And, of course, the pad design is being affected accordingly.

For synchronizing the active players in the app, I have implemented a timing mechanism based on "ticks". I have a global variable (called "tick") that is only relevant if at least one player is active.

At the same time, the first player in a sequence is turned on, the tick is initialized to 1. Then, on every second its value increases by one until its value is equal to the number of seconds of one complete loop cycle, then, it sets back to 1.

Then, when an additional player is turned on, its audio is only turned on, on the next loop cycle (next time "tick" is being reset to 1).

When the "**stop all active players**" button is clicked – it collects all the active players, backs them up to lastActivePlayersList variable, and then turns off each one of them.

"**play last active players**" button – iterates

on the players that were backed up to lastActivePlayersList and turn on

each one of them.

**Recording Bonus**

My idea for the recording part was to record each user interaction in the app and the players he interacted with. Then, "playing" the recording is actually just repeating the recorded actions in the same sequence and the same timing.

For that reason, recording involves one more timing mechanism that is not limited to the length of the loop cycle. The practice is the same, I keep one more variable (recordingTick) that is also based on the "ticks" mechanism (as explained above).

Then when a user action is recorded (by calling addRecordingAction function) the current recordingTick and the players that he interacted with are being added to the queue recordActionsQueue with the current value of recordingTick.

So, recordActionQueue is an array of the following format:

[{tick: recordingTick, players: [player1, player2,…]}]

By decision, starting a record session stops all active players.

"**Play session**" – imitate the behavior described in the recordActionsQueue.