Ideas:
Chess engine
Physics sim
Pathfinding sim
Boids
File compression
Audio - different plugins (echo, bitcrunch, compression)

Outline:
The program will have a jframe where you can select an audio file (wav, maybe others) and an effect. Once you select the effect you will be able to press a button labeled “apply” where it will change the data and create a new file. Each effect class will also have an undo function, and I will have an arraylist of the effects applied so that you can get your original audio back. The audio effect class will be abstract with the abstract method apply effect. I will make the project using the terminal first, and then port it to jframe.

Unknowns: Arraylist or custom effectlist class?

All effect ideas:
Easy: Echo/delay/reverb (what is the difference between these), bitcrusher, normalize, reverse
Medium: Low/high pass filter, distortion, speed change, fade in / out, tremolo
Hard: equalizer

Throughout:
Each audio effect will be its own class.

Step 1:
Learn the format of a wav file, read it, make a clip and then turn it back into a file.
Classes: WavFileReader, WavFileWriter, AudioClip

Step 1.5:
Research plugins and mp3. Will I have mp3 support, which plugins are best to add?

Step 2:
First plugin: Change volume and pitch of the file
Classes: AudioEffect

Step 3:
Echo/delay/reverb? Effect

Step 4:
Bitcrush

Step 5:
Distortion

Day 1:
Wav file header:

Byte number	Value	Actual information in english
1 - 4	”RIFF”	A string literal “RIFF” Marks the file as a wav
5 - 8	File size (integer)	How big the file is (minus 1-8)
9 - 12	“WAVE”	A string literal saying “WAVE”
13 - 16	“Fmt “	A string literal saying “fmt “ with space / null
17 - 20	16	16 The length of the data listed above
21 - 22	Type of format	1 for uncompressed audio
23 - 24	Number of channels	1 = mono, 2 = stereo
25 - 28	Sample rate	44100 hz (cd quality) - how many “snapshots” of the analogue wave are taken/played per second
29 - 32	Byte Rate	(Sample Rate * BitsPerSample * Channels) / 8. Equates to how many bytes per second of audio
33 - 34	Block align	NumChannels × BitsPerSample / 8. 1 = 8 bit mono. 2 = 8 bit stereo / 16 bit mono. 4 = 16 bit stereo. Block align = number of bytes per sample frame Sample frame = one sample per channel at a given point in time
35 - 36	Bits per sample	Cd quality = 16. Number of bits used to represent amplitude (loudness)
37 - 40	”data”	A sting literal saying “data” represents the beginning of data

41 - 44	File size (data)	Indicates how many bytes of audio follow this (not including header)

http://www.topherlee.com/software/pcm-tut-wavformat.html
https://i.sstatic.net/Q9xGC.jpg
https://docs.fileformat.com/audio/wav/

Day 2 / 3 / 4 - Writing a wav file:
All Java integer types (byte, short, int, long) are signed, but a WAV file header expects numbers to be written as unsigned bytes. Even though Java doesn't provide unsigned primitive types, we can still write the correct unsigned values because Java stores all numbers in two's complement binary, and the lowest 8 bits of any number always match their unsigned 8-bit representation.

-255 = 1 and -254 = 2 when you compare their first byte

To do this, we use the & operator and compare our value to "0xFF" (hexadecimal form of 255, we could also use the decimal form "255" however using the hexadecimal format makes it more obvious.)
When we use the bitwise & operator, it compares the binary values of the byte and extracts the lowest 8 bits, therefore, using this method we get only the first 8 bits of our value.
EXAMPLE: 8 in a java short is 0000 0000 0000 1000, negative numbers in java are stored using the following steps.

First, invert all numbers, therefore, our number becomes 1111 1111 1111 0111.
Next, add 1, 1111 1111 1111 1000. This equals -8

So to convert this into unsigned numbers, we just take the first (least significant / lowest) byte, which in this case is equal to 248.
The WAV file header uses little endian values so we have to write them backwards. We do this by writing the smallest byte value and then use the bitwise right shift to write the next more significant byte.

Therefore, I will create little endian (LE) / unsigned writer methods that write bytes in LE and convert them into unsigned integers. There will be one for int and one for short. I will also need to write my strings into bytes, however this is much easier as there is a java method for that.

Day 5 - Reading a wav file and first effect:
FileInputStream we can read all the bytes in a wav file, add all the data to an array list, edit them, then recreate the wav. The header data will be added to a byte array as it is a set length.

First effect: volume: I will index through each short of data, convert it to int and then multiply it by a set value. Then I will clamp the value to the max value of a short before converting it back.

Day 6 - Creating pitch effect:
I added a pitch/speed effect by changing the sample rate of the audio clip. This makes the file read each sample faster.

Day 7 - Reverse effect and organization:
Added a reverse effect
General organization of files (merged file reader and file writer into one file manager file, and added export folder for edited audios, changed util folder to ByteManager and added byte reading function to clear up the audioclip file)
feature where each effect will add the effect to the file name (ex: soundeffect_reversed_spedUp.wav).
Researched the difference between Echo/delay/reverb:

Delay- a single repeat of a sound at x amount of time later
Echo- multiple repeats of the same sound, each quieter than the last
Reverb - way too hard to implement, thousands of tiny echos happening simultaneously (could fake it by layering multiple short delays with different times and decays)

Day 8 - J Frame:
I created a simple J frame interface, it's very ugly and almost unusable, however it works.
I got bored of the J frame and made a tremolo effect.

Day 9 - J Frame 2 and bug turned feature:
Completely restarted the J Frame, I found another way to select files that is much more visually appealing.
I had a bug that the audio file kept in memory was not the original, but was instead the most recent version (with all the effects). This meant that every time you added an effect, it would add it on top of all the previous ones.
I realized this made it a lot easier to chain effects (the original idea behind this project) but I wanted to be able to add effects without chaining them. To do this I created a method called copy. This keeps a copy of the original file, and just makes a copy of it every time we add an effect.
To keep the best of both worlds, I added a check box named “chain effects” when this was on, the copy method is not used.