LION SYNTH

1. Las clases
2. Colecciones de datos
3. Guardado y carga de datos
4. Ventanas (layouts)

LAS CLASES

* Ventanas / frames
  + SynthWindow
  + OscilloscopeWindow2
  + Keyboard
  + Keyboard2
* Las clases para la capa de negocio
  + Envelope
  + Filter
  + Function
  + LionSynth (Router)
  + Mixer
  + Oscillator
* Las clases enumeradas
  + FilterType
  + MusicalNote
  + WaveShape
* La clase abstracta
  + SingleSignalProcessor

**Ventanas***SynthWindow*:

Para evitar concentrar todo el código en el código en el constructor de la ventana, hay algunas partes modularizadas. Si los nombres no resultan lo suficientemente descriptivos, aquí una breve explicación:  
*guiComponentsDeclaration*: como dice el nombre, aquí es donde se declaran los componentes.  
*addComponentsToWindow*: añade los componentes a la ventana.   
*addAllComponentsListeners*: llama a otros métodos encargados de crear los listeners para los osciladores, envelope, etc.

Los únicos Listeners que hay dentro del constructor son para los menu items de guardar y cargar presets, botón de reproducir sonido y el menu item que abre la ventana del osciloscopio.

Algo interesante a tener en cuenta es que la clase de la ventana contiene todas las referencias a los componentes internos del sintetizador. Así pueden ser modificadas fácilmente desde los Listeners declarados para la ventana. El enrutador solo recibirá los punteros a estos objetos, puesto que tendríamos que copiar entero cada objeto cada vez que es actualizado.

Un ajuste curioso con el que cuenta la ventana es poder cambiar de *look and feel*. Esta característica puede ser accedida desde la barra de menús, en el menú de Skin. Ahí hay varias para escoger. Al seleccionar una, se crea (o sobrescribe) un archivo (con un Print Stream y un File Output Stream), *lionSynth.init*, que contiene los valores con los que se inicializa el sintetizador. Por el momento solo especifica la skin que está usando el usuario. Como se trata de un archivo sin formato, puede ser modificado por el usuario sin problema, aunque solo algunos valores son válidos. No se cambia la skin en caso de que no lo sean. Para cambiar el *look and feel*, se llama al método *applySkin* antes de crear la ventana, el cual cambia el *look and feel* leyendo el *lionSynth.init* con un File Input Stream y un Scanner.

*OscilloscopeWindow2*:

Esta ventana hará las de osciloscopio.

En la ventana principal, cada vez que se pulsa el botón de “play”, la señal de salida del enrutador se renueva. Cuando pulsamos al botón de refrescar onda o abrimos la ventana por vez primera, se copia dicha salida del enrutador a un atributo de la clase de la ventana del osciloscopio, *array*, la cual se dibuja en un panel en la derecha de la ventana.

El dibujado de dicha onda no es algo complejo. La señal *per se* es un conjunto de valores entre 1 y -1 que indican cuál es la altura de la onda en cada momento a lo largo del tiempo. Con un bucle *for* podemos dibujar líneas que conectan cada punto de la onda con el siguiente e ir dibujándolas todas a lo largo del eje. A todos los valores le sumaremos la mitad de la altura de la pantalla para que estén ajustados al centro.

Por último, para que la onda se ajuste a la altura de la pantalla lo que hago es buscar el valor más alto (en volumen) dentro de la señal de entrada y multiplicar dicho valor por la mitad de la altura de la pantalla (cada mitad corresponde al lado positivo y negativo de la onda). El valor de cada punto será multiplicado por este valor cuando vaya a ser dibujado. Como la señal toma valores de 1 a -1, tenemos que redimensionarla a la fuerza, y la ventaja de usar este método es que se verá con una misma altura dos ondas de distinto volumen.

**Frames:**

*Keyboard* se trata de una clase experimental que pinta un teclado en un panel. Está sin implementar.

Keyboard2 es la clase hija de *Panel* que finalmente ha servido de teclado. Será sustituida por *Keyboard* en cuanto esté lista. Está creada como clase a parte, para no tener que definir todos sus componentes como parte de la ventana principal. Es una clase sencilla que contiene doce botones de color blanco o negro, imitando las teclas del piano. De este modo es muy sencillo operar con ella.

**Clases enumeradas:**

*FilterType*:

Aunque el filtro no funcione, sí que está definido y cuenta con una propiedad que determina por dónde se empiezan a cortar frecuencias. Esto lo señala el tipo de filtro seleccionado y los hay de muchos tipos, de los cuales he seleccionado los cuatro fundamentales.

*WaveShape*:

Similar al tipo de filtro, es una clase enumerada creada para limitar el tipo de ondas que sirven de entrada a los osciladores. Los hay de cinco tipos.

*MusicalNote*:

El rol de esta clase es asociar los nombres de las notas a sus valores en Hz, solo en la 5 octava, pues el resto pueden ser calculados a partir de ellas. Por cuestiones de comodidad, las he implementado tanto en el formato anglosajón (C, D, E, …) como en el latino (Do, Re, Mi, …). Para que la clase enumerada pueda funcionar como constantes, hay que declararlos como atributos de la clase.  
Cada elemento de la clase guardará su nombre, su frecuencia en Hz y la frecuencia en Hz de su sostenido. Las notas sostenidas tomarán el valor de un semitono hacia arriba (la tecla contigua a la derecha). De este modo, todas las notas guardan un valor para el sostenido y no solo las notas que cuentan con uno.

La clase cuenta con constructores, pues son condición necesaria para asociar valores. Para ahorrarme trabajo, creé un constructor que copiaba una nota y redefinía el nombre, para crear la versión latina a partir de la anglosajona sin tener que copiar valores.

**La clase abstracta:**

La única clase abstracta del programa, que la he creado para cumplir los requisitos del trabajo, se llama *SingleSignalProcessor* y la implementan aquellas clases de la capa de negocio que son capaces de procesar (tomar una señal de entrada y transformarla) una única señal de sonido en formato de array de doubles.

**Clases de la capa de negocio:**

*Disclaimer: las capas de negocio cuentan con aspectos técnicos que explico, explicaciones que pueden ser leídas para una mejor comprensión del funcionamiento, pero explicaciones que, al fin y al cabo, pueden ser omitidas.  
  
Envelope*:

*Explicación omisible:  
Así como un oscilador de baja frecuencia (no he implementado ninguno) simplemente modifica un valor del sintetizador a lo largo del tiempo, un envelope lo hace en relación a la pulsación de la tecla. En este sinte solo contamos con un envelope para el volumen, de modo que será este el que cambie con respecto a la pulsación. A alguien en universo se le ocurrió que el recorrido que sigue el volumen de un instrumento a lo largo del tiempo puede ser dividido en varias fases. Estas fases son cuatro: el attack, el decay, el sustain y el release.  
El attack corresponde con el inicio del sonido. Determina cuánto tiempo tardará el sonido en llegar desde el silencio hasta el volumen máximo. ¿Por qué el máximo? Porque el creador pensó que era mejor así. Hay otros modelos que permiten cambiar ese aspecto, este no. El sustain es el volumen en el que se queda mientras el usuario mantiene pulsada una tecla. En una flauta, equivale a seguir soplando. En un piano siempre es cero. El release corresponde al final del sonido; es el tiempo que tarda el volumen en regresar al silencio después de que el usuario haya soltado la tecla (o dejado de soplar). El release siempre va después del sustain. El decay puede que sea el más extraño de todos, pero cobra sentido cuando el sustain es cero. El decay es el tiempo que tarda el sonido en ir del attack al sustain. Cuando el sustain es cero, el release en inexistente, y el sonido saltará a cero lo que dure el decay, de tal modo que este será el parámetro que indique cuándo deja de sonar el sonido, tomando el rol de release.  
Ejemplo: el piano tarda poco en empezar y no podemos mantener la tecla pulsada para que suene mucho más tiempo. Esto implica que tiene un attack corto, un sustain nulo y un decay (tomando el rol de release) que varía según cuánto se esté pisando el pedal del piano.   
Un violín tarda un poco más que el piano en empezar, no golpea tanto, pero acaba más rápido y puede mantenerse el volumen siempre y cuando se siga tocando. Attack largo y release corto, sustain variable.*

El envelope cuenta con los cuatro atributos que lo definen: *attack, decay, sustain y release.*  
En su método *process*, se tomará una señal de entrada, la cual será procesada y se devolverá al ser llamado al método. Esto le convierte en un *SingleSignalProcessor*.

Lo primero que se hará dentro de este método será calcular las longitudes en samples de las distintas fases del envelope. Para ello, tomo los valores de *attack, decay y release* de la instancia, que están en segundos, y los multiplico por la frecuencia de sampleo (sample rate). En caso de que la duración de cualquiera de las fases exceda la duración del sample, se irá recortando cuántos samples corresponde a cada fase hasta ajustarse, dando prioridad al *attack*.

Luego, se abren cuatro hilos que procesan cada una de las fases por separado y de forma distinta. Para entender bien cómo funciona cada hilo internamente es necesario entender el funcionamiento de un envelope. Más allá del funcionamiento individual de cada uno, lo que harán por norma general es multiplicar por un valor de atenuación los samples del array de entrada de la fase que tengan asignada. Si la fase debe cambiar a lo largo del tiempo, se llamará a una función lineal que irá variando la atenuación para que se ajuste a su propósito individual.

No se dará por terminado el procesamiento de la señal de entrada hasta que los cuatro hilos hayan terminado.

*Filter:*

Esto requeriría una explicación gorda por aquí.

El filtro no está terminado, no procesa de forma adecuada las señales. No lo voy a explicar innecesariamente.

*Function:*

Una clase similar a *Math* de Java. Contiene funciones útiles para la ejecución de los distintos procesos dentro de la capa de negocio. Hay cuatro funciones que sirven para generar las ondas del oscilador: *pulse, triangle, saw* y *square*.

Por otro lado, tenemos una función que pretende atenuar de forma realista el sonido (pues nuestra percepción del volumen sigue una curva y no una recta). Sin embargo, es algo chapucera y me queda pendiente corregirla.

Luego, hay dos funciones lineales, que se diferencian por recibir distintos argumentos. Es tan sencillo como una ecuación de la recta implementada para las fórmulas de dos puntos y la estándar. Las usa el envelope.

Por último, una función para los sliders, porque tampoco siguen una función lineal. La explico con el HashMap.

*LionSynth:*

El LionSynth toma el papel de enrutador. Recibe al ser declarado punteros a los distintos componentes del sintetizador y éste se encarga de unirlos de forma adecuada, llevando las señales de los osciladores, a un mixer que las junta, al envelope y luego a la salida.

Posteriormente creé, en un intento de hacer que el sintetizador funcionara en tiempo real, un método, *processStream*, que coordina de forma distinta el enrutamiento, y funcionó, pero resultó un infierno y no suena como debería. El método estándar para enrutar las señales es *process*, llamado por el botón de “play”. Este método siempre se llama en un hilo nuevo y solo puede ser accedido de forma coordinada.

*Mixer:*

No es una *singleSignalProcessor*, porque procesa dos señales. Lo que hace es unir dos señales de entrada en una, sumando sus valores. Es realmente simple.

*Oscillator:*

Es el encargado de generar las señales. Cuenta con una serie de atributos típicos de los osciladores como la octava a la que suena (octave), la nota a la que suena con respecto de la que le indica el sinte (note y fine), la forma de la onda (waveform), el detune (no usado), el volumen (tampoco usado) y la frecuencia.

Internamente, el oscilador toma sus atributos como instancia y genera una onda con dicha configuración. El único valor que toma como argumento es la duración del sample a generar. Se genera la onda mediante el método *genSample*. También hay una versión alternativa para el proyecto fallido de hacer el sintetizador funcionar en tiempo real, *genStreamSignal*.El método tiene un case para decidir a qué método delegará el generar el sample, en función de la onda seleccionada.

Por norma general, estos llenan un array de samples de la duración que se les indica y con los valores propiciados por la función de la onda correspondiente.

COLECCIONES DE DATOS

La funcionalidad más interesante dentro del sintetizador para el *HashMap* la tiene la clase de *Function,* en *adsrSlider*. Para los sliders del envelope, como no quería que siguieran una función lineal como el resto, lo que hice fue mapear una serie de puntos por los que pasaría la función. Luego, en lugar de calcular una función que se ajustara, lo cual es imposible, cada vez que quisiera saber el valor de un punto buscaría el valor más cercano por encima y por debajo dentro del mapa y calcularía su posición relativa con respecto a ellos.

GUARDADO Y CARGA DE DATOS

Los presets:  
Un *preset* es un archivo que guarda el estado actual de un sintetizador digital al completo, para poder compartir, vender, etc. sonidos creados con dicho sintetizador. Es una herramienta muy popular en el mundo de la síntesis, aparentemente sencilla, aunque me ha dado algún que otro quebradero de cabeza al implementarla. No funciona entre sintetizadores.

La opción de guardar y cargar un preset está localizada en la barre de menús, dentro del menú de “File”. Como los nombres indican, “Load preset from file…” cargará el estado del sintetizador desde un archivo y “Save preset to file” guardará el estado actual del sintetizador en un archivo.

Evidentemente, los *menu items* de carga y guardado de presets cuentan con un *Action Listener* asociado, donde se ejecutarán sus sendos códigos.

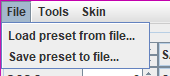


Imagen 1: barra de menús del sintetizador con el menú de File desplegado.

Dependiendo de la opción que se elija, se abrirá una ventana de JFileChooser (de la API de Swing) en el directorio en el que se encuentra el sintetizador para que el usuario escoja el archivo sin tener que escribir la ruta al completo. Una vez se ha escogido el archivo, se guarda la dirección absoluta del archivo en una variable de tipo File, clase de la API de IO de Java.

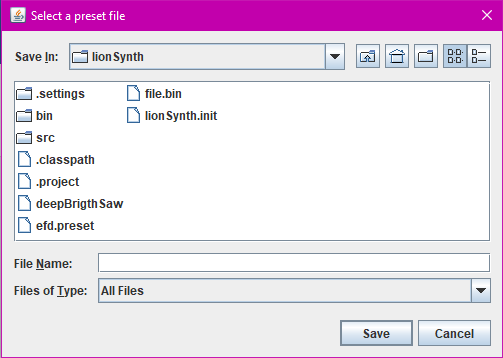
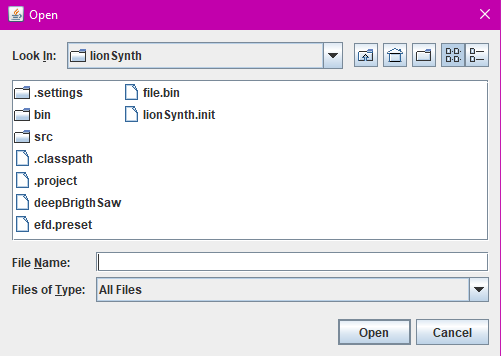


Imagen 2: JFileChooser para guardar un preset Imagen 3: JFileChooser para abrir un preset

Una vez obtenida dicha dirección, los procesos divergen:

En el guardado de datos, se abrirá un ObjectOutputStream que guardará el estado del router (LionSynth). Tanto esta como las clases que contiene implementan *Serializable*, requisito para que el ObjectOutputStream funcione.  
Al mismo tiempo, se abre un FileOutputStream, al que se le pasa como argumento la ruta del archivo obtenida por el JFileChooser y se escribe el enrutador en dicho archivo. Por supuesto, estas líneas de código están rodeadas de un *try catch* para evitar que la ejecución se rompa por cualquier motivo.

La carga de datos fue algo más complicada. Se abre un ObjectInputStream y un FileInputStream para leer el archivo indicado por el JFileChooser. Luego, se guarda el objeto leído, el enrutador, en una nueva instancia y se van copiando las propiedades de los componentes, el envelope principal, el mixer, los osciladores, etc. a los ya existentes en el programa. Cabe recordar que el enrutador usa las referencias a estos objetos, no los copia, para que sea posible realizar cambios sobre los osciladores y resto de componentes sin tener que llamar al enrutador.  
La verdadera lata de este asunto fue refrescar los componentes en la GUI, de hecho, los sliders del ADSR están sin actualizar. Más que nada porque para sacar la posición en la que debiera estar el slider tendría que sacar la función inversa que uso para dar valores al ADSR. Y eso es una lata muy gorda. A parte de eso, para no meter tanto código en el *Action Listener* de la función de cargar un preset, traslado el refresco de estos componentes a un método, *copyValuesToComponents()*.

VENTANAS (LAYOUTS)

Para este proyecto, he creado dos ventanas. La ventana principal, *SynthWindow*, y una ventana que muestra el osciloscopio. Sobre la ventana principal están todos los componentes y respectivos *Listeners*. La ventana principal estaba dividida en varios layouts, diseño *responsive* del cual sé un poco pues en bachiller vi un poco de HTML y CSS. Dentro del propio código, donde los componentes se establecen como atributos de clase, los he identado de tal manera que sea intuitivo cómo se han distribuido las layouts. Recomiendo abrir el código, pues una vez visto ni siquiera hace falta leer esta sección del documento.

El layout principal es un estándar BorderLayout. He creado un contenedor por cada una de estas regiones que uso del BorderLayout. Estas son *northPanel*, que va en el centro puesto que el “norte” lo ocupa la barra de menús, y el *southPanel*, que sí que va al “sur”.  
El *northPanel* está dividido en dos paneles, el *oscPanel1 y el* oscPanel2. Dentro de los dos paneles están distribuidos los cuatro osciladores, con sus respectivas *Combo boxes*, *spinners* y etiquetas.  
El *southPanel* está divido también en dos. La parte sur es el teclado (que es un JFrame por sí solo) y en la parte norte están acomodados el panel del envelope, el mixer y las opciones generales del sinte.