Curso de Supercollider

Taller de Audio del Centro Multimedia

Principiantes

Ernesto Romero y Ezequiel Netri

Romero, Ernesto y Netri, Ezequiel.
Curso de Supercollider principiantes.
México D.F.: Centro Multimedia, 2008.

La Clase 8 (Buffer), está basada en una clase de Sergio Luque.

Primer revisión, febrero 2012

Transcripción a pdf, Hernani Villaseñor

http://cmm.cenart.gob.mx/tallerdeaudio
email: tallerdeaudio@gmail.com

Clase 1: Introducción y sintaxis

s.boot;

http://www.audiosynth.com/
http://supercollider.sourceforge.net/
http://cmm.cenart.gob.mx/tallerdeaudio

Introducción
¿Que es SuperCollider?
Un editor de texto: escribimos líneas de código que contienen instrucciones para sintetizar sonidos y manipularlos. Este código es guardado como texto. El editor no sólo contiene las funciones típicas de cualquier procesador de textos básico, como copiar, pegar, buscar y reemplazar; sino que también tiene otras funciones que son muy útiles al programar y compilar.

Un lenguaje de programación: orientado a objetos, basado en el lenguaje SmallTalk.

Es software open source, es gratuito y construido por la comunidad.

Es muy portable.

Es un lenguaje interpretado para sintesis de audio.

Primeros pasos

Cuando arrancamos el programa aparece una ventana con información, conocida como *Post Window*. En ésta se imprimirá información útil, como los mensajes de error o los resultados de nuestros códigos.

La primera información que vemos es el resultado de la inicialización del programa y de la

compilación de su librería.

Necesitamos saber algunas cosas antes de comenzar

1. Para correr lineas de código:
    Usa enter, no return (MacOSX)
    ctrl Return (Windows)
    ctrl E (Linux-gedit)

2. Antes de crear cualquier sonido debemos de prender alguno de los dos servidores.

3. Para detener el sonido
    command + punto [manzana .] (MacOSX)
    alt + punto (Windows)
    esc (Linux-gedit)

4. Para la Ayuda seleccionamos y tecleamos
    command+D [manzana + D] (MacOSX)
    F1 (Windows)
    ctrl U (Linux-gedit)

//evalua las siguientes líneas, una por una

```
1 + 1
2 * 2
5 / 2
2**2
2 ** 3
1<2
0.0323>0.2
"hola"=="ola"
```

Para poder producir sonido necesitamos prender un servidor. Tenemos dos distintos servidores predefinidos: interno y local.

El **servidor interno** corre en el mismo proceso que la aplicación SuperCollider, es interno al programa y por eso mismo tiene ciertas ventajas, producto de la mayor comunicación entre los dos.

El **servidor local** corre en la misma máquina que la aplicación SuperCollider, pero es un programa diferente: 'scsynth'. La ventaja de usarlo es que en caso de que el servidor se caiga, la aplicación seguirá corriendo y viceversa.

También es posible crear más servidores y que estos estén en distintas computadoras comunicadas vía internet o ethernet.

Sólo es necesario prender el servidor una vez por sesión. Hay dos formas de hacerlo, a través de las ventanas que los representan (abajo a la izquierda) o usando código.

```
s=Server.local
s.boot
s.quit

s=Server.internal
s.boot
s.quit
```

UGENs

Los UGens o *Unit Generators* son objetos que producen algún tipo se señal como SinOsc, Pulse, Saw, o LFTri. Al conectar varios de ellos creamos lo que se conoce como *patch*.

Sus nombres siempre empiezan con mayúscula y pueden ser de dos tipos:

Audio Rate o .ar

Los UGens que reciben el mensaje .ar corren a velocidad de audio: 44100 muestras por segundo. Hay que mandar el mensaje .ar a los UGens cuando sean parte de la cadena de audio que será escuchada.

Control Rate o .kr

Los UGens que reciben el mensaje .kr corren a velocidad de control. Producen una muestra por cada 64 muestras hechas por un UGen a velocidad de audio. Es por esto que los UGens de control son más baratos, computacionalmente hablando, que sus contrapartes a velocidad de audio.

Los UGens de control los usamos como moduladores, esto es, como señales que le dan forma a una señal de audio.

```
s=Server.local
s.boot
```

{ }.play es la forma más simple para generar sonido, muy útil para probar código rápidamente, pero no muy conveniente para la construcción de piezas.

```
{    }.play //se usa para el servidor local
{    }.scope //se usa para el servidor internal
```

Veamos los siguientes osciladores básicos:
Todos los oscliadores comprenden una serie de argumentos ordenados que representan los parámetros que indican el comportamieto del mismo. En otras palabras es el medio por el cual le comunicas al UGen lo que quieres que haga.

Los argumetos básicos de casi cualquier oscilador son :

frecuencia, es decir la altura de una nota.
fase,que por el momento la dejaremos en 0.
amplitud, el volumen de una nota.

Estos argumentos son generales para todos los Ugens, aunque existen UGens con argumentos exclusivos.

```
{SinOsc.ar (700,0,0.1) !2}.play // senoide
{Pulse.ar (700,0.2,0.1) !2}.scope // cuadrada
{Saw.ar (700,0.1) !2}.scope  // diente de sierra
{LFTri.ar (700,0,0.3) !2}.scope  // triangular
```

Osciladores

SinOsc es un oscilador de onda senoidal. Funciona con .ar y con .kr. Sus argumentos son frecuencia, fase, multiplicación y adición. Su sintaxis es como sigue:

```
SinOsc.ar (frecuencia, fase, mul, add)
```

frecuencia: Ciclos por segundo o Hertz. Una frecuencia alta nos dará una frecuencia aguda. Una frecuencia baja nos dará una nota grave. El rango auditivo del ser humano es de 20 Hz a 20 kHz. El *default* es 440 Hz que es la nota La índice 5.

fase: Punto del ciclo en el que queremos que inicie el oscilador. Se especifíca en radianes y su rango es de 0 a 2pi. El *default* es 0 que es el inicio del ciclo.
mul: Número por el cual multiplicamos la señal del oscilador. Generalmente se identifica con el volumen o amplitud del sonido siendo 0 el mínimo y 1 el máximo recomendado. El *default* es 1 dejando la señal sin alterar.
add: Número que se le suma a la señal del oscilador. Observar que a la señal se le aplica primero el mul y luego el add. El *default* es 0 dejando la señal sin alterar.

Ejemplos:

{SinOsc.ar}.play // El SinOsc con los argumentos de default o sea frecuencia=440, fase=0, mul=1, add=0.

{SinOsc.ar(100)}.play // Con una frecuencia grave. Si no tienes audífonos o bocinas de buena calidad no la vas a oír por que las bocinas de la computadora no pueden hacer sonar frecuencias graves.

{SinOsc.ar(300,0,0.5)}.play // Una frecuencia media, con un volumen (mul) medio.
{SinOsc.ar(3000,0,0.1)}.play // Una frecuencia aguda con un volumen bajo.
{SinOsc.ar(13000,0,0.9)}.play // Una frecuencia muy agudacon un volumen alto.

Pulse es un oscilador de onda cuadrada.

Funciona solo con .ar. Sus argumentos son frecuencia, ancho de banda, mul y add. Su sintaxis es como sigue:

Pulse.ar (frecuencia, ancho de banda, mul, add)

frecuencia: Ciclos por segundo o Hertz. El default es de 440.

ancho de banda: El ancho del valle de la onda (la parte de abajo o el período de apagado). El rango esta comprendido por el intervalo abierto (0, 1) o por un número mayor que cero y menor que uno. El *default* es 0.5. Al cambiar este argumento estamos cambiando el tímbre de la señal.

mul: Número por el cual multiplicamos la señal del oscilador. Generalmente se identifica con el volumen o amplitud del sonido siendo 0 el mínimo y 1 el máximo recomendado. El *default* es 1 dejando la señal sin alterar.

add: Número que se le suma a la señal del oscilador. Observar que a la señal se le aplica primero el mul y luego el add. El default es 0 dejando la señal sin alterar.

Ejemplos:

{Pulse.ar}.play // El Pulse con los argumentos de default.
{Pulse.ar(100)}.play // Con la frecuencia grave. En este caso si se oye en las bocinas de la computadora por la característica del tímbre que es rico en armónicos que son frecuencias más agudas.
{Pulse.ar(100,MouseX.kr(0.1,1))}.scope // La misma frecuencia pero con un ancho de pulso menor. Observar cómo cambia el timbre pero la nota sigue siendo la misma.
{Pulse.ar(100,0.9,0.1)}.play // La misma frecuencia pero con el ancho de pulso mayor. La amplitid baja. Nótese que el timbre no varía ya que el ancho de pulso es simétrico con su eje de simetría en 0.5 de tal modo que 0.1=0.9, 0.2=0.8, 0.7=0.3, etc.

LFTri es un oscilador de onda triangular.

Funciona con .ar y .kr. Sus argumentos son frecuencia, mul y add. Su sintaxis es como sigue:

LFTri.ar (frecuencia, fase, mul, add)

frecuencia: Ciclos por segundo o Hertz. El default es de 440.
mul: Número por el cual multiplicamos la señal del oscilador. Generalmente se identifica con el volumen o amplitud del sonido siendo 0 el mínimo y 1 el máximo recomendado. El default es 1 dejando la señal sin alterar.

add: Número que se le suma a la señal del oscilador. Observar que a la señal se le aplica primero el mul y luego el add. El default es 0 dejando la señal sin alterar.

Ejemplos:

{LFTri.ar}.play // El Pulse con los argumentos de default.
{LFTri.ar(150)}.scope // Con la frecuencia grave. En este caso si se oye en las bocinas de la computadora por la característica del tímbre que es rico en armónicos que son frecuencias más agudas.
{LFTri.ar(1000,0.1)}.play // Con la frecuencia aguda y poca amplitud.

Saw es un oscilador de onda de sierra.

Funciona solo con .ar. Sus argumentos son frecuencia, mul y add. Su sintaxis es como sigue:

Saw.ar(frecuencia, mul, add)

frecuencia: Ciclos por segundo o Hertz. El *default* es de 440.
mul: Número por el cual multiplicamos la señal del oscilador. Generalmente se identifica con el volumen o amplitud del sonido siendo 0 el mínimo y 1 el máximo recomendado. El *default* es 1 dejando la señal sin alterar.
add: Número que se le suma a la señal del oscilador. Observar que a la señal se le aplica primero el mul y luego el add. El *default* es 0 dejando la señal sin alterar.

Ejemplos:

{Saw.ar}.play // El Saw con los argumentos de default.
{Saw.ar(350)}.scope // Con la frecuencia grave. En este caso si se oye en las bocinas de la computadora por la característica del tímbre que es rico en armónicos que son frecuencias más agudas.
{Saw.ar(10000,0.1)}.play // Con la frecuencia aguda y poca amplitud.

Ruidos

{WhiteNoise.ar(0.19)!2}.scope //señal aleatoria que contiene todas las frecuencias y todas ellas tienen la misma potencia

{PinkNoise.ar(0.9)!2}.scope //señal aleatoria que sus contenido de energia por frecuencia disminuye en 3 db por octava.

{BrownNoise.ar(0.8)!2}.scope //señal aleatoria que sus contenido de energia por frecuencia disminuye en 6 db por octava.

{Dust.ar(100)!2}.scope//generador de impulsos aleatorios

Abre el archivo SC2-examples_1.rtf que se encuentra en el directorio supercollider/examples/demostrations/SC2-examples_1.rtf

Ejercicios

Vamos a ejercitar una de las técnicas más sencillas para crear nuevos timbres. Esta técnica tiene como principio el utilizar dos operadores básicos: la multiplicación y la suma.

Suma

Al aplicar una suma entre dos osciladores obtenemos un nuevo timbre en donde se puede escuchar que las características de los objetos que participan no pierden su "personalidad". Esta técnica tiene relación con lo que se conoce como sintesis aditiva. Observar que las amplitudes de las señales se suman. Las suma de las amplitudes no deben de superar 1.0.

{SinOsc.ar (2000,0,0.2)+WhiteNoise.ar(0.1) !2}.play
{SinOsc.ar(2000,0,0.2)!2}.play
{WhiteNoise.ar(0.1)!2}.play

Multiplicación

Al aplicar una multiplicación entre dos osciladores obtenemos un timbre complejo que esta determinado por diferentes factores que analizaremos con el transcurso del taller. Esta

técnica tiene relación con lo que se conoce como Amplitud Modulada y Modulación de Anillo

```
{SinOsc.ar (2000,0,0.7)*WhiteNoise.kr(0.8) !2}.scope
{WhiteNoise.ar (0.8)*SinOsc.kr(2000,0,0.7) !2}.play
{WhiteNoise.kr (0.8)*SinOsc.ar(2000,0,0.7) !2}.play
```

Completa los espacios en blanco

1

```
{-----.ar(1360,0,-----)*WhiteNoise.kr(3)}.scope
```

2

```
{-----.ar(60,0.5)*-----.kr(780,0,0.34)}.scope
```

3

```
{-----.ar(-----,0,0.5)*-----.kr(240,0,0.5,0.5)}.scope
```

4

```
{------.ar(0.3)*-------.kr(3,0,0.9)}.scope
```

5

```
{Pulse.ar(--,--,0.5)*----.kr(100,---)}.scope
```

6

```
{-----.ar(30,0.3)*--------.kr(0.6)}.scope
```

7

```
{LFTri.ar(30,0.1,0.3)*--------.kr(0.6)*------.kr(10,0,0.4,0.6)}.scope // !!!!!!
```

8

```
{----.ar(302,0.3)*--------.kr(10.6,0,0.6)*------.kr(0.710)}.scope
```

9

```
{SinOsc.ar(----,0,0.5)*SinOsc.kr(----,0,0.5,0.5)}.scope
```

10

```
{----.ar(---,0.4)*WhiteNoise.kr(---)*----.ar(1200,0.4)*SinOsc.kr(---,0,0.6)}.scope
```

11

```
{(------.ar(---)*----.kr(0.2))+Saw.ar(---,0.041)}.scope
```

12

{(Pulse.ar(40,0.01,0.3)*-----.kr(---,0,0.4,0.5)*WhiteNoise.kr(---))+----.ar(---,0.4)}.scope

13

{((--------.ar(0.5)*Dust.kr(----))+PinkNoise.ar(---))*SinOsc.kr(----,0,0.4,0.5)}.scope

14

{Saw.ar(---,0.3)+Saw.ar(---,0.3)}.scope

15

{((Saw.ar(460,0.3)+Pulse.ar(462,---,0.3))*Dust.kr(---))+LFTri.ar(---,0.1,----)}.scope

16

{----.ar(14000,0,0.9)*Dust.kr(-----)}.scope

17

{----.ar(70,0,---)*Dust.kr(----)}.scope

18

{SinOsc.ar(-----,0,0.3)+-----.ar(----,0,0.3)+SinOsc.ar(----,0,0.2)+------.ar(50,0,0.7)}.scope

19

{LFTri.ar(---,0,0.5)*SinOsc.kr(----,0,0.7)}.scope

20

{-----.ar(130,0.2)+-----.ar(100,0,0.3)+-------.ar(0.031)}.scope

Clase 2: Filtros

Soluciones a la tarea 1

```
{SinOsc.ar(1360,0,0.6)*WhiteNoise.kr(0.3)}.scope
{Saw.ar(60,0.5)*SinOsc.kr(780,0,0.34)}.scope
{SinOsc.ar(1000,0,0.5)*SinOsc.kr(240,0,0.5,0.5)}.scope
{BrownNoise.ar(0.3)*SinOsc.kr(3,0,0.9)}.scope
{Pulse.ar(10,0.1,0.5)*LFTri.kr(100,0.5)}.scope
{Saw.ar(30,0.3)*WhiteNoise.kr(0.6)}.scope
{LFTri.ar(30,0.1,0.3)*WhiteNoise.kr(0.6)*SinOsc.kr(10,0,0.4,0.6)}.scope
{Saw.ar(302,0.3)*SinOsc.kr(10.6,0,0.6)*WhiteNoise.kr(0.710)}.scope
{SinOsc.ar(100,0,0.5)*SinOsc.kr(2,0,0.5,0.5)}.scope
{Saw.ar(12000,0.4)*WhiteNoise.kr(0.3)*Saw.ar(1200,0.4)*SinOsc.kr(40,0,0.6)}.scope
{(PinkNoise.ar(0.2)*WhiteNoise.kr(0.2))+Saw.ar(40,0.041)}.scope
{(Pulse.ar(40,0.01,0.3)*SinOsc.kr(30,0,0.4,0.5)*WhiteNoise.kr(0.8))+Saw.ar(10,0.4)}.scope
{((WhiteNoise.ar(0.5)*Dust.kr(10))+PinkNoise.ar(0.9))*SinOsc.kr(100,0,0.4,0.5)}.scope
{Saw.ar(30,0.3)+Saw.ar(70,0.3)}.scope
{((Saw.ar(460,0.3)+Pulse.ar(462,0.4,0.3))*Dust.kr(10))+LFTri.ar(42,0.1,0.2)}.scope
{SinOsc.ar(14000,0,0.9)*Dust.kr(300)}.scope
{LFTri.ar(70,0,0.3)*Dust.kr(100)}.scope
{SinOsc.ar(45,0,0.3)+SinOsc.ar(40,0,0.3)+SinOsc.ar(100,0,0.2)+SinOsc.ar(50,0,0.3)}.scope
{LFTri.ar(30,0,0.5)*SinOsc.kr(1000,0,0.7)}.scope
{Saw.ar(130,0.2)+SinOsc.ar(100,0,0.3)+WhiteNoise.ar(0.031)}.scope
```

Filtros

Un filtro es un sistema que, dependiendo de algunos parámetros, realiza un proceso de discriminación de una señal de entrada obteniendo variaciones en su salida. Notar que en todos los casos de los filtros que vamos a ver el primer argumento es la señal que se desea filtar y el segundo argumento es la frecuencia de corte. La frecuencia de corte en los filtros

HPF y LPF tiene un significado ligeramente distinto que en el filtro BPF. Los tres filtros que veremos a continuación pueden funcionar tanto con .ar como con .kr. La condición al usarlos es que la señal que se quiera filtrar tenga el mismo rate que el filtro.

HPF - High Pass Filter (Filtro Pasa Altas)

Es aquel que permite el paso de frecuencias desde una frecuencia determinada hacia arriba, sin que exista un límite superior especificado. Esta frecuencia determinada es la frecuencia de corte. Por ejemplo, si determinamos que sea 700 Hz la frecuencia de corte dejaremos pasar todas las frecuencias mas altas que 700 Hz.

Argumentos: entrada, frecuencia de corte, multiplicación y adición.

Sintaxis: HPF.ar (entrada, frecuencia de corte, multiplicación, adición)
entrada: La señal que queremos filtrar. Tiene que tener .ar.
frecuencia de corte: Frecuencia en Hertz a partir de la cual se permitirá el paso de frecuencias mas altas.
multiplicación: Número por el cual multiplicamos la señal del filtro. Generalmente se identifica con el volumen o amplitud del sonido siendo 0 el mínimo y 1 el máximo recomendado. El default es 1 dejando la señal sin alterar.
adición: Número que se le suma a la señal del filtro. Observar que a la señal se le aplica primero el mul y luego el add. El default es 0 dejando la señal sin alterar.

LPF - Low Pass Filter (Filtro Pasa Bajas)

Es aquel que permite el paso de frecuencias bajas, desde la frecuencia 0 hasta una frecuencia determinada. Esta frecuencia determinada es la frecuencia de corte. Por ejemplo, si determinamos que sea 200 Hz la frecuencia de corte dejaremos pasar todas las frecuencias mas bajas que 200 Hz. Recordemos que no hay frecuencias menores a 0 Hz.

Argumentos: entrada, frecuencia de corte, multiplicación y adición.

Sintaxis: LPF.ar(entrada, frecuencia de corte, multiplicación, adición)
entrada: La señal que queremos filtrar. Tiene que tener .ar.

frecuencia de corte: Frecuencia en Hertz a partir de la cual se permitirá el paso de frecuencias mas bajas.

multiplicación: Número por el cual multiplicamos la señal del filtro. Generalmente se identifica con el volumen o amplitud del sonido siendo 0 el mínimo y 1 el máximo recomendado. El default es 1 dejando la señal sin alterar.

adición: Número que se le suma a la señal del filtro. Observar que a la señal se le aplica primero el mul y luego el add. El default es 0 dejando la señal sin alterar.

BPF - Band Pass Filter (Filtro Pasa Banda)

Es aquel que perimte el paso de frecuencias contenidas dentro de un determinado rango o banda, comprendido entre una frecuencia inferior y otra superior. La distancia entre estas frecuencias determina el ancho de banda. La frecuencia que está en el centro de esta distancia es la frecuencia de corte. Por ejemplo, si determinamos que sea 1000 Hz la frecuencia de corte y 200 Hz el ancho de banda podemos saber cual es el rango de frecuencias que dejaremos pasar usando la siguiente fórmula:

cota inferior = frecuencia de corte - ancho de banda/2

cota superior = frecuencia de corte + ancho de banda/2

La cota superior es la frecuencia límite superior y la cota inferior es la frecuencia límite inferior.

Si sabemos cuáles son las cotas inferior y superior que queremos entonces podemos obtener el ancho de banda y la frecuencia de corte con la siguiente fórmula:

ancho de banda = cota superior - cota inferior

frecuencia de corte = cota inferior + ancho de banda/2

{BPF.ar(WhiteNoise.ar, 300, 100)}.scope

BPF no es el único filtro que utiliza un ancho de banda. En este filtro y en todos los demás de este tipo el ancho de banda no se puede escribir directamente como un argumento. En vez de ancho de banda estos filtros tienen como argumento rq.

q = frecuencia de corte / ancho de banda.
Por lo tanto el reciproco de q = 1/q = ancho de banda / frecuencia de corte = rq
Argumentos: entrada, frecuencia de corte, rq, multiplicación y adición.

Sintaxis: BPF.ar (entrada, frecuencia de corte, rq, multiplicación, adición)

entrada: La señal que queremos filtrar.
frecuencia de corte: Frecuencia en Hertz que determina el centro de la banda de nuestro filtro.
rq: recíproco de q, es decir, ancho de banda / frecuencia de corte.
multiplicación: Número por el cual multiplicamos la señal del filtro. Generalmente se identifica con el volumen o amplitud del sonido siendo 0 el mínimo y 1 el máximo recomendado. El default es 1 dejando la señal sin alterar.
adición: Número que se le suma a la señal del filtro. Observar que a la señal se le aplica primero el mul y luego el add. El default es 0 dejando la señal sin alterar.

Algunos ejemplos.

// Aquí un WhiteNoise (Ruido Blanco) es filtrado por un filtro pasa altas dejando pasar frecuencias arriba de los 7030 Hz

```
{HPF.ar(WhiteNoise.ar(0.1), 7030)}.scope

// Aquí lo contrario: pasan las frecuencias debajo de los 7030 Hz
{LPF.ar(WhiteNoise.ar(0.1), 7030)}.scope

// El tercer argumento del filtro pasa banda es el reciproco de Q.
{BPF.ar(WhiteNoise.ar(0.1), 7030, 703/7030)}.scope
```

En este ejemplo tenemos ancho de banda=700 Hz y frecuencia de corte= 7030 Hz. O sea, 703/7030 = 0.1

Por lo tanto rq = 0.1

A veces es mas rápido escribir el número decimal que el quebrado. Veamos entonces como queda sustituyendo del ejemplo anterior:

```
{BPF.ar (WhiteNoise.ar (0.1), 7030, 0.1)}.scope

{BPF.ar (WhiteNoise.ar(0.1), 7030, 1)}.scope // Aquí tenemos otro valor para el rq.
```

Si tenemos que 1 = ancho de banda / frecuencia de corte = rq
Entonces sabemos que ancho de banda = frecuencia de corte = rq
Y si frecuencia de corte = 7030 Hz
Entonces rq = 7030 / 7030 Hz
Por lo tanto sabemos que la línea de código anterior se puede escribir también de la siguiente forma:

```
{BPF.ar(WhiteNoise.ar(0.1), 7030, 7030/7030)}.scope
```

Hay señales que se prestan a ser filtradas y otras que no. Por ejemplo, una senoide (SinOsc) no se presta a ser filtrada porque solo contiene una frecuencia. No existe nada más arriba ni más abajo de esa frecuencia para ser retirado. Los ruidos, que son un conjunto complejo de

frecuencias son los que dan resultados más notables al ser filtrados.

Control con el Mouse

Una forma de controlar los argumentos de los UGens en SuperCollider es a través de las Clases MouseY, MouseX y MouseButton. Estas clases son UGens de control por lo que trabajan con el mensaje kr. Los argumentos principales de los UGens MouseY y MouseX son el valor mínimo y el valor máximo que queremos obtener al mover nuestro *mouse*. En MouseButton los argumentos son el valor cuando el mouse no está apretado y cuando si.

En SuperCollider para Windows se utiliza JMouseY, JMouseX y JMouseButton. Al final de tu línea de código agregar
JmouseBase.makeGUI, separado por;
Ejemplos:

```
MouseX.kr(0,100) // obtengo números desde 0 cuando el mouse esta hasta la izquierda hasta 100 cuando el mouse esta  en el extremo derecho.

MouseY.kr(0,100) // obtengo números desde 0 cuando el mouse esta hasta arriba hasta 100 cuando el mouse esta  hasta abajo.
```

Notar que en Y el máximo es abajo y el mínimo arriba. En X el máximo es a la derecha y el mínimo a la izquierda.

```
MouseButton.kr(0,1) // Cuando no esta apretado tengo el valor 0 y cuando si el valor 1.

{SinOsc.ar(MouseY.kr(1000,100))}.scope // Puedo hacer un barrido desde la frecuencia 100 Hz cuando el mouse esta abajo hasta 1000 Hz cuando el mouse esta hasta arriba

{SinOsc.ar(400,0,MouseX.kr(0,1))}.scope // Amplitud 0 cuando el mouse esta en el extremo izquierdo hasta 1 cuando el mouse esta en el extremo derecho
```

```
{SinOsc.ar(MouseX.kr(1000,100),0,MouseY.kr(0,1))}.scope // Mezclando los dos ejemplos anteriores con MouseY y MouseX

{Pulse.ar(MouseY.kr(10000,20),MouseX.kr(0.001,0.5),MouseButton.kr(0,1))}.scope
```

Los mismos ejemplos de filtros usando la clase MouseY y MouseX

```
{ HPF.ar(WhiteNoise.ar(0.1), MouseY.kr(10000,100))}.scope
{ LPF.ar(WhiteNoise.ar(0.1), MouseX.kr(10000,100))}.scope
{ BPF.ar(WhiteNoise.ar(0.1), MouseY.kr(10000,100), 0.05)}.scope
{ BPF.ar(WhiteNoise.ar(0.5), 130, MouseX.kr(0.05,1.5))}.scope
{ BRF.ar(WhiteNoise.ar(0.1), MouseY.kr(10000,100), 1)}.scope
{ BRF.ar(WhiteNoise.ar(0.1), 7030, MouseX.kr(0.05,1.5))}.scope
```

Los mismos ejemplos pero para SuperCollider en Windows

```
{SinOsc.ar(JMouseY.kr(1000,100))}.jscope;JMouseBase.makeGUI
{SinOsc.ar(400,0,MouseX.kr(0,1))}.jscope;JMouseBase.makeGUI
{SinOsc.ar(MouseX.kr(1000,100),0,MouseY.kr(0,1))}.jscope;JMouseBase.makeGUI
{Pulse.ar(MouseY.kr(10000,20),MouseX.kr(0.001,0.5),MouseButton.kr(0,1))}.Jscope;JMouse
Base.makeGUI
```

Los mismos ejemplos de filtros usando la clase MouseY y MouseX

```
{ HPF.ar(WhiteNoise.ar(0.1), JMouseY.kr(10000,100))}.jscope;JMouseBase.makeGUI
{ LPF.ar(WhiteNoise.ar(0.1), JMouseX.kr(10000,100))}.jscope;JMouseBase.makeGUI
{ BPF.ar(WhiteNoise.ar(0.1), JMouseY.kr(10000,100), 0.05)}.jscope;JMouseBase.makeGUI
{ BPF.ar(WhiteNoise.ar(0.5), 130, JMouseX.kr(0.05,1.5))}.jscope;JMouseBase.makeGUI
{ BRF.ar(WhiteNoise.ar(0.1), JMouseY.kr(10000,100), 1)}.jscope;JMouseBase.makeGUI
{ BRF.ar(WhiteNoise.ar(0.1), 7030, JMouseX.kr(0.05,1.5))}.jscope;JMouseBase.makeGUI
```

Tarea 2

```
{----.ar(BrownNoise.ar(0.5),160)}.scope
{HPF.ar(----.ar(10,0.75)*WhiteNoise.ar(1),10000)}.scope
{Saw.ar(10,0.75)*----.ar(WhiteNoise.ar(1),10000)}.scope
{----.ar(Saw.ar(2,0.6),3000,100/3000)+BPF.ar(----.ar(3,0.4),----,50/500)}.scope
{LFTri.ar(1,0,1)*LPF.ar(----.ar(30,0.1,1),1000)}.scope
{Pulse.ar(117,----.kr(0.5,0.01),0.5)}.scope
{----.ar(MouseX.kr(60,800),0,0.6)}.scope
{----.ar(----.ar(100),MouseY.kr(15000,160),60/MouseY.kr(15000,160))}.scope
{Pulse.ar(----.kr(5,20))----(SinOsc.ar(10000,0,0.6)----SinOsc.ar(90,0,0.6))}.scope
{Pulse.ar(MouseY.kr(110,90),----,0.5)*----.ar(100,0.1,0.5)}.scope
```

Clase 3: Envolventes, canales y lenguaje

Existen algunas notas señaladas dentro del texto. Estas pueden o no ser leídas sin afectar la comprensión de la clase. Sin embargo para el que tenga o quiera tener un conocimiento de música le ayudarán a implementar algunas estructuras musicales dentro del código.

Soluciones a la tarea 3

```
{LPF.ar(BrownNoise.ar(0.5),160)}.scope
{HPF.ar(Saw.ar(10,0.75)*WhiteNoise.ar(1),10000)}.scope
{Saw.ar(10,0.75)*HPF.ar(WhiteNoise.ar(1),10000)}.scope
{BPF.ar(Saw.ar(2,0.6),3000,100/3000)+BPF.ar(Saw.ar(3,0.4),500,50/500)}.scope
{LFTri.ar(1,0,1)*LPF.ar(Pulse.ar(30,0.1,1))}.scope
{Pulse.ar(117,MouseY.kr(0.5,0.01),0.5)}.scope
{SinOsc.ar(MouseX.kr(60,800),0,0.6)}.scope
{BPF.ar(Dust.ar(100),MouseY.kr(15000,160),60/MouseY.kr(15000,160))}.scope
{Pulse.ar(MouseX.kr(5,20))*(SinOsc.ar(10000,0,0.6)+SinOsc.ar(90,0,0.6))}.scope
{Pulse.ar(MouseY.kr(110,90),0.1,0.5)*Pulse.ar(100,0.1,0.5)+SinOsc.ar(70,0,0.3)+SinOsc.ar(13000,0,0.3)}.scope
```

Envolventes

La envolvente es la manera en que se despliega un sonido en función del tiempo y la amplitud. Estamos hablando de qué tan fuerte suena nuestro timbre a medida que pasa el tiempo. Por ejemplo una envolvente percusiva es como la de un tambor o un piano, en donde el sonido comienza muy fuerte e inmediatamente se va escuchando mas quedito. Los elementos de la envolvente más comunmente identificados son:

ataque (attack): que tan rápido alcanza nuestro sonido su punto de máxima amplitud.
decaimiento (decay): que tan rápido alcanza nuestro sonido su punto de estabilidad o sostenimiento.

sostenimiento (sustain): cuanto tiempo está nuestro sonido en el punto de estabilidad en el que no sube ni baja de amplitud.
liberación (release): el tiempo que tarda el sonido en llegar del punto de sostenimiento a la amplitud cero.

Estos términos son conocidos en inglés como *Attack, Decay, Sustain* y *Release*. Algunas veces se utilizan sus siglas para referirse a la envolvente que posee estos elementos: ADSR. No todas las envolventes tienen todos estos elementos. En algunas envolventes se necesita menos información para crear su contorno. Podemos imaginar por ejemplo una envolvente que sólo tenga ataque y liberación (*attack, release*). En SuperCollider para generar una envolvente recurrimos a la clase EnvGen. Las envolventes pueden ser pensadas como una secuencia de números que se despliega en el tiempo. Esta secuencia puede ser usada para varios prósitos que pueden involucrar una forma de onda o una modulación. Por el momento nuestras envolventes no generarán sonido, solo determinan la manera en que su amplitud se desarrolla en el tiempo, asi que EnvGen trabajará con el rate de control .kr.

EnvGen.kr EnvGen.kr (envolvente,gate,doneAction:2)
Genera una envolvente que puede dar forma a la amplitud de nuestro sonido.

envolvente: En el argumento envolvente colocamos alguna de las muchas envolventes que posee SuperCollider en la clase Env.

Env.adsr (ataque, decaimiento, volumen, relajamiento)
Envolvente de duración indeterminada para sonidos sostenidos. El segundo argumento es de volumen, no de duracion.

Env.perc (ataque, liberación)
Envolvente de ataque percusivo, argumentos en segundos. Duración determinada por la suma del *attack* más el *release*.

Ejemplo:

Env.perc (0.01,1).plot // Aquí el attack dura 0.01 segundos y el release 1 segundo 0.01 + 1 = 1.01. Observar que con el mensaje .plot podemos ver la gráfica de la envolvente

Env.asr (ataque, volumen, decaimiento)
Envolvente de duración indeterminada para sonidos sostenidos. El segundo argumento es de volumen, no de duración.

Env.asr(0.01,1,1).plot // Se tardará una centésima de segundo en alcanzar su amplitud máxima, que es 1 como lo indica el segundo argumento. Una vez alcanzada se quedará ahí hasta que nosotros le indiquemos cuando generar el release que se tardará 1 segundo en alcanzar el cero.

gate 1 abre la envolvente, 0 la cierra. Default 1. Las envolventes de duración determinada como Env.perc no necesitan cerrarse.

doneAction una acción que se realiza cuando la envolvente ha finalizado. doneAction: 2 elimina el Synth del servidor.

Ejemplo de Env.asr

```
SynthDef("prueba", {|gate|
Out.ar(0,Pulse.ar(15)*EnvGen.kr(Env.asr(0.01,1,3),gate,doneAction:2))}).send(s)

a=Synth("prueba", ["gate", 1]) // Lo prenden
a.set("gate", 0) // Lo apagan
```

Para aplicar la envolvente a un sonido multiplicamos la envolvente por el sonido. Ejemplos:

```
{Saw.ar(40)*EnvGen.kr(Env.perc(0.1,0.1),doneAction:2)}.scope

{Saw.ar(40)*EnvGen.kr(Env.asr(1,1,4),Line.kr(1,0,2),doneAction:2)}.scope
```

Observar que en el argumento gate hemos colocado un UGen Line que genera una línea de números que va desde el 1 al 0 en 2 segundos. Esto nos abre y cierra la envolvente automáticamente (ver el *Help* de Line).

Canales

El sonido análogo o digital puede salir por uno o varios canales. Nosotros estamos acostumbrados a escuchar la música en dos canales que suenan en una bocina cada uno. Esto es conocido como estereofonía y es por eso que al aparato de sonido de nuestro auto o casa le llamamos "el estéreo".
Existen aparatos que nos permiten sacar el sonido por más de dos canales. Estos aparatos son conocidos como tarjetas o interfaces de audio. Las hay de 2, 4 u 8 canales por lo menos. Además estas interfaces se pueden conectar entre sí sumando la cantidad de canales. En SuperCollider existen varias clases que nos ayudan a trabajar con los canales de audio por donde queremos que salga nuestro sonido. Aquí veremos 2: Out.ar y Pan2.ar.

Out.ar Out.ar (canal,señal)
Saca el sonido por un canal específico. Ese canal específico define un punto de partida u *offset* a partir del cual se va a distribuir el sonido.

canal: 0 = izq, 1 = der. 3, 4, 5,...multicanal
señal: cualquier oscilador que puede estar multiplicado por una envolvente.

Ejemplos:

```
{Out.ar(0,Saw.ar(40)*EnvGen.kr(Env.perc(0.01,1),doneAction:2))}.scope  // izquierda
```

```
{Out.ar(1,Saw.ar(40)*EnvGen.kr(Env.perc(0.01,1),doneAction:2))}.scope  // derecha
```

Pan2.ar Pan.ar (señal,posición)
Distribuye el sonido entre dos canales consecutivos conservando su potencia. Es decir, que

no suena más fuerte cuando esta en los dos canales al mismo tiempo ni más quedito cuando está solo en uno o en otro. Si el Pan2 esta dentro de un Out los canales consecutivos en los que se distribuyen se cuentan a aprtir del *offset* del Out.

señal: cualquier oscilador o generador de sonido
posición: -1 izquierda, 1 derecha y con todo el intervalo continuo extrapolando el sonido entre los dos canales o bocinas.

```
{Pan2.ar(Pulse.ar(100,0.01),MouseX.kr(-1,1))}.scope
{Pan2.ar(Pulse.ar(100,0.01),-0.7)}.scope
{Pan2.ar(Pulse.ar(100,0.01),0)}.scope // En medio.
{Pan2.ar(Pulse.ar(100,0.01),0.3)}.scope
{Pan2.ar(Pulse.ar(100,0.01),1)}.scope

{Out.ar(0,Pan2.ar(Dust.ar(1000),0))}.scope // distribuye la señal entre el canal 0 y 1
{Out.ar(1,Pan2.ar(Dust.ar(1000),0))}.scope // distribuye la señal entre el canal 1 y 2. Si no tenemos una interfase de audio que nos permita expandir los canales solo se va a escuchar al canal 1. Solo tenemos 2 canales en nuestras computadoras que el Collider reconoce como el 0 y el 1.
```

Lenguaje

```
.midicps // convierte un número de código MIDI a su equivalente en frecuencia en Hertz.
.cpsmidi // convierte una frecuencia en Hertz a su equivalente en código MIDI.
```

El código MIDI para designar las notas es:

60=Do (índice 5 o central)
61=Do # o Re b
62=Re
63=Re # o Mi b

64=Mi

65=Fa

66=Fa # o Sol b

67=Sol

68=Sol # o La b

69=La

70=La # o Si b

71=Si

72=Do (índice 6. Una octava arriba del Do índice 5)

Si queremos convertir el código MIDI 60 (Do) en frecuencia en Hertz lo hacemos mandándole el mensaje .midicps (cps=ciclos por segundo).

60.midicps

69.midicps

Para el inverso aplicamos el mensaje cpsmidi

261.6255653006.cpsmidi

440.cpsmidi

.midiratio // convierte intervalos en razones o quebrados

Los intervalos para usar con el mensaje midiratio son expresados en cantidad de semitonos. Está relacionado con el método de Forte para los pitch class sets:

0 = unísono

1 = segunda menor

2 = segunda mayor

3 = tercera menor

4 = tercera mayor

5 = cuarta justa
6 = tritono
7 = quinta justa
8 = sexta menor
9 = sexta mayor
10 = séptima menor
11 = séptima mayor
12 = octava

Los números negativos denotan intervalos descendentes. Entonces para aplicar este mensaje mandamos el mensaje . midiratio al intervalo que deseamos obtener y lo multiplicamos por una frecuencia fundamemntal que nosotros damos. Ejemplo:

```
440 * 3.midiratio // nos da una tercera menor a partir de La.
440 * -5.midiratio // nos da una carta descendente a partir de La.
```

Array

Un conjunto de elementos ordenados. Se escriben dentro de corchetes [ ] y se separan por comas.

```
['hola', 'hi', 'salud', 'ciao']
[0,1,2,3,4]
[0,1,2,3,4].choose
[60,62,64,65,67].midicps // Nota 1
[ 261.6255653006, 293.66476791741, 329.62755691287, 349.228231433,
391.99543598175 ].cpsmidi
```

Tarea 3

```
{----.ar(0,SinOsc.ar)}.scope;
{Pan2.ar(WhiteNoise.ar,----)}.scope;
```

```
{Out.ar(1,Saw.ar(100)*EnvGen.kr(--Env--.perc(0.01,2),doneAction:----2))}.scope;
{----.ar(WhiteNoise.ar,
[100,200,400,1000,1500,5000].----,0.1)*EnvGen.kr(Env.perc(0.01,0.5),doneAction:2)}.play;
{Pan2.ar(SinOsc.ar([60,64,67,72].choose.----),[----,----].choose)}.play;
{[LPF,HPF].----.ar(BrownNoise.ar,800)}.play;
```

Nota 1

Un poco de música para el que quiera.

Diferentes escalas usando el código MIDI.

```
[60,62,64,65,67,69,71,72].midicps // un array con las notas de la escala mayor
[60,62,63,65,67,68,70,72].midicps // un array con las notas de la escala menor natural
[60,62,64,66,68,70,72].midicps // la escala de tonos enteros
[60,62,63,65,66,68,69.72].midicps // La escala simétrica tono, 1/2tono
[60, 61, 63, 64, 66, 67.72, 71].midicps // La escala simétrica 1/2tono, tono
```

Las mismas escalas expresadas en intervalos y usando midiratio

```
[ 0, 2, 4, 5, 7, 9, 11, 12 ].midiratio // un array con las notas de la escala mayor
[ 0, 2, 3, 5, 7, 8, 10, 12 ].midiratio // un array con las notas de la escala menor natural
[ 0, 2, 4, 6, 8, 10, 12 ].midiratio // la escala de tonos enteros
[ 0, 2, 3, 5, 6, 8, 9, 12 ].midiratio // La escala simétrica tono, 1/2tono
[ 0, 1, 3, 4, 6, 7, 12, 11 ].midiratio // La escala simétrica 1/2tono, tono
```

Notar que para convertir las escalas expresadas en MIDI en el primer conjunto de arrays a las escalas expresadas en intervalos del segundo conjunto basta con restarles 60. Veamoslo en el primer ejemplo de las escala mayor.

```
[60,62,64,65,67,69,71,72]-60 == [ 0, 2, 4, 5, 7, 9, 11, 12 ]
```

Clase 4: Variables y SynthDef

Soluciones a la tarea 4

```
{Out.ar(0,SinOsc.ar)}.scope;
{Pan2.ar(WhiteNoise.ar,0)}.scope;
{Out.ar(1,Saw.ar(100)*EnvGen.kr(Env.perc(0.01,2),doneAction:2))}.scope;
{HPF.ar(WhiteNoise.ar,
[100,200,400,1000,1500,5000].choose,0.1)*EnvGen.kr(Env.perc(0.01,0.5),doneAction:2)}.sco
pe;
{Pan2.ar(SinOsc.ar([60,64,67,72].choose.midicps),[-1,1].choose)}.play;
{[LPF,HPF].choose.ar(BrownNoise.ar,800)}.play;
```

Variables

Son espacios virtuales de memoria que sirven para guardar información.

Notas:

- El concepto de variable es inherente a los lenguajes de programación.
- Facilitan la organización del código permitiendo una clara y legible escritura.
- Son identificadas por medio de un nombre que generalmente se relaciona con el objeto al cual queremos igualarlo. Este nombre lo definimos nosotros.
- No tenemos un número determindo de variables, podemos declarar cuantas sean necesarias.
- La variable no varía. Una vez guardada dentro de una estructura, como por ejemplo dentro de un SynthDef, no se podrá manipular su valor.

Para crear nuestras variables es necesario respetar algunos pasos:

1- Escribimos var (abreviatura de variable) dejamos un espacio y empezamos a declarar las

variables. Finalizamos la enumeración de las variables con un punto y coma (;).

Ejemplo: compila el código y observa la post.

```
//bien
(
var leon, dos;
leon = 1;
dos = 2;
leon + dos
)

//mal, observa que contesta la post
(
leon = 1;
dos = 2;
leon + dos
)
```

2- Separamos las variables por medio de comas (,).
3- Para finalizar la declaración de nuestras variables colocamos un punto y coma(;)
4- Las variables que contengan un solo caracter (ej: a,b,c,d) no necesitan ser declaradas. Recuerda que la letra 's' esta designada para los servidores, por ende es recomendable no utilizarla.
5- Las variables empiezan con minúscula.

Ejemplo de variables de un solo caracter:

```
(
a = 1;
x = 2;
a + x
```

```
)
```

Ejemplo en donde incluimos la operación con las variables dentro de un array.

```
(
var leon, dos;
leon = 1;
dos = 2;
[leon + dos,'dos variables']
)
```

También podemos darle valor a la variables en el lugar donde la declaramos.

```
(
var leon = 1, dos = 2;
leon + dos
)
```

Para escribir nombres largos y estrafalarios, utilizamos el guión bajo en lugar del espacio. También se pueden usar mayúsculas.

```
(
var un_numero_mayor_a_diez = 15, un_numero_mayor_a_20 = 43;
un_numero_mayor_a_diez + un_numero_mayor_a_20
)
```

```
(
var unNumeroMayorADiez = 15, unNumeroMayorA20 = 43;
unNumeroMayorADiez + unNumeroMayorA20
)
```

Ahora con señales de audio

```
(
{
var ruido, senoide;
ruido = WhiteNoise.ar(0.2);
senoide = SinOsc.ar(1000,0,0.3);
ruido + senoide
}.scope
)
```

```
(
{
var ruido = WhiteNoise.ar(0.5), senoide = SinOsc.ar(10, 0, 0.3);
var env;
env = EnvGen.kr(Env.perc(0, 0.2), Impulse.kr(4));
senoide * ruido * env
}.scope
)
```

Manzana + F = find (encontrar en Linux que onda), con lo que ustedes podrán encontrar con mucha facilidad palabras o símbolos que esten buscando dentro de un código. Una vez encontradas pueden ser sustituidas de una sola vez.

```
(
{
var oscilador, envolvente, impulso;
impulso = Impulse.kr(MouseX.kr(1, 10));
oscilador = LFTri.ar(200*[1, MouseY.kr(0.98, 1.0124)],0 , 0.53);
envolvente = EnvGen.kr(Env.perc(0.01, 1), impulso);
oscilador * envolvente;
```

```
}.scope
)
```

Sin variables el código resulta algo borroso, por ejemplo:

```
{EnvGen.kr(Env.perc(0.01, 1), Impulse.kr(MouseX.kr(1, 10)))*LFTri.ar(200 * [1, MouseY.kr(0.98, 1.0124)],0 ,0.53)}.scope
```

SynthDef

Es una definición de sonido (creado por nosotros) o una definición de síntesis que puede ser invocado y manipulado independientemente de otros. Un SynthDef es el método por el cual creamos nuestros sonidos en SuperCollider.

SuperCollider utiliza la definición de Synth 'SynthDef' como un template para la creación de *synth node* que es una unidad productora de sonido en el servidor.

Un SynthDef tiene una sintáxis específica que comprende dos partes fundamentales:
el nombre del SynthDef que se escribe como un string o un símbolo (ej :"prueba" o \prueba) por el cual podremos manipular el synth independientemente de otros synth y un UgenGraphFunc = una función en donde se especifíca la intercomunicación entre variables y argumentos.

Ejemplo de la clase anterior:

```
(
SynthDef("prueba", {|gate,frecuencia=15|
Out.ar(0,Pulse.ar(frecuencia)*EnvGen.kr(Env.asr(0.01,1,3),gate,doneAction:2))}).send(s)
)

a=Synth("prueba",[\gate,1])
a.set(\gate,0)
```

La sintáxis de SynthDef comienza con el objeto SynthDef

```
(
SynthDef("prueba", //el SynthDef debe de llevar un tag (nombre) que lo identifique, en este caso "prueba".
{|gate|      //argumento, es el lugar destinado para los parámetros que queremos interpretar
             //los argumentos son también espacios virtuales de memoria que sirven para guardar información pero esta, a diferencia de las variables es creada para ser modificada
Out.ar(0,          //canal de salida, 0 y 1
Pulse.ar(15)*EnvGen.kr(Env.asr(0.01,1,3),gate,doneAction:2)) //salida
}).send(s)         //se envia el Synth al servidor designado por default con la letra 's'
)

(
SynthDef("prueba", {|gate=1|
        var sen,env;
        sen = Pulse.ar(15, 0.2, 0.1);
        env = EnvGen.kr(Env.asr(0.01, 1, 3),gate, doneAction:2);
        Out.ar(0, sen * env)
        ).send(s)
)

(
SynthDef("prueba",{arg gate=1;
        var sen,env;
        sen=Pulse.ar(15,0.2,0.1);
        env=EnvGen.kr(Env.asr(0.01,1,3),gate,doneAction:2);
        Out.ar(0,sen*env)
        }).send(s)
)
```

```
(
{
var sen,env;
sen = Pulse.ar(15, 0.2, 0.5);
env = EnvGen.kr(Env.perc(0, 0.1),Impulse.kr(2));
sen * env
}.play
)
Synth("prueba")
```

Una vez creada la estructura del SynthDef es necesario invocarlo para que suene, el SynthDef no suena por si solo. Para esto tenemos dos manera de hacerlo, una de ella es la llamada Object Style.

1.- Object Style

```
Synth(nombre, [\argumento1, valor1, ... \argumentoN, valorN], target, addAction)
```

```
(
Synth(                          // Utilizamos el método new de la clase Synth
        "prueba",               // Un string que especifique el nombre del SynthDef
                                // a utilizar para crear el nuevo synth.

        [\gate, 1, \frecuencia, 20], // Un Array opcional que especifique los valores iniciales
                                // para los argumentos del SynthDef estos valores son
                                // mandados en parejas, el nombre del argumento como
                                // symbol y su valor:[\nombre1, valor1, \nombre2, valor2]

        s,                      // El target: el grupo donde será creado el nodo de este
                                //synth.
                                // si especificamos al servidor como target, nuestro synth
```

```
                                    // será creado en el grupo default del servidor, que es el
                                    // 1

            \addToTail              // el addAction: especificamos en qué lugar del grupo
                                    // será creado nuestro synth
        )
)
```

Para facilitar la invocación del SynthDef podemos utilizar las instrucciones más básicas dejando por default el target y la addAction:

Ejemplo

```
a = Synth("prueba")
a = Synth("prueba")
a = Synth("prueba", [\gate, 1]) // Lo prenden
a.set("gate", 0) // Lo apagan
```

Como notaran igualando a la variable 'a' un SynthDef que se llama "prueba" para luego modificar sus argumentos por el método *.set*.
Los argumentos a los que nos referimos son los que creamos en el SynthDef y esta es la manera en la que cambiamos su valor.

Hay ciertos componentes de esta estrucutura de los SynthDef que pueden escribirse de otro modo:

```
(
SynthDef(\prueba, //el tag cambia de comilla a diagonal (cambia de string a symbol)
{arg gate;  // los argumentos pueden enumerarse después de la palabra arg y concluye con
            // punto y coma(;).similar a las variables
Out.ar(0,   //canal de salida, 0 y 1
```

```
        Pulse.ar(15)*EnvGen.kr(Env.asr(0.01,1,3),gate,doneAction:2)) //salida
        }).send(s)          //se envia el synth al servidor designado por default con la letra s
)

a=Synth(\prueba, [\gate, 1]) // las comillas se cambian por diagonales
a.set(\gate, 0) // Lo apagan

(
SynthDef(\mi_primer_synth,{|frecuencia, amplitud|
        var sen, env, trig;
        trig = Impulse.kr(2);
        sen = Saw.ar(frecuencia * [0.988889, 1.011], amplitud);
        env = EnvGen.kr(Env.perc(0.01, 0.5), trig);
        Out.ar(0, sen * env)
        }).send(s)
)

a=Synth(\mi_primer_synth)
a=Synth(\mi_primer_synth,[\frecuencia,200,\amplitud,0.3])
a.set(\frecuencia,12)
a.free
```

SynthDef con envolvente percusiva más un argumento para el triger

```
(
SynthDef(\mi_primer_synth,{|frecuencia=100,amplitud=0.6,frecuenciatrig=2|
        var sen, env, trig;
        trig = Impulse.kr(frecuenciatrig);
        sen = Saw.ar(frecuencia * [0.988889, 1.011], amplitud);
        env = EnvGen.kr(Env.perc(0.01, 0.5), trig);
        Out.ar(0, sen * env)
```

```
        }).send(s)
)

a = Synth(\mi_primer_synth)
a.set(\frecuenciatrig, 1)
a.set(\frecuenciatrig,5.rrand(10),\frecuencia,101.rrand(1820))
a.free
```

SynthDef con envolvente asr

```
(
SynthDef(\mi_segundo_synth,{|frecuencia=100, amplitud=0.6, gate=1|
        var sen,env,trig;
        sen=Saw.ar(frecuencia*[0.988889,1.011],amplitud);
        env=EnvGen.kr(Env.asr(3.6,1,4.5),gate,doneAction:2);
        Out.ar(0,sen*env)
        }).send(s)
)

a=Synth(\mi_segundo_synth)
a.set(\frecuencia,10)
a.set(\gate,0)
```

En este ejemplo mostramos como podemos crear diferentes Synth a partir de un SynthDef.

```
(
SynthDef(\hola,{|frec=430,amp=0.2,gate=1,trig=10|
        var sen,env;
        sen=SinOsc.ar(frec*[1,1.01],0,amp);
        env=EnvGen.kr(Env.perc(0,0.1),Impulse.kr(trig));
        Out.ar(0,sen*env)
        ).send(s)
)
```

```
a=Synth(\hola)
a.set(\frec,1700)
a.free

b=Synth(\hola,[\frec,1240])
b.set(\frec,800)
b.free

c=Synth(\hola,[\frec,900])
c.free
```

Cuando copilamos la línea de código: a = Synth (\hola)
La post nos devuelve: Synth("hola" :1000)

Esta línea nos dice el nombre del SynthDef utilizado por el Synth y el nodo en donde éste último se encuentra (1000). El número de nodo (nodeID) fue seleccionado automáticamente por la clase Synth, ésta simplemente buscó un número de nodo que estuviese libre.

Nodo: un objeto con el que se puede establecer comunicación, se encuentra en un árbol de nodos manejado por el servidor de síntesis. El árbol define el orden de ejecución de todos los Synths. Hay dos tipos de nodos: groups y synths. Todos los nodos llevan un número entero por nombre (nodeID).

Tarea

```
(
SynthDef(\----,{|---,---,---|
	var ----,----;
	----=SinOsc.ar(----,0,----);
	---=EnvGen.kr(Env.perc(0,---),doneAction:2);
	---.ar(----,----*----)
```

```
        }).----(s)
)

(
------(\----,{|---,---,---,----|
        var ----,----,----;
        ---=-----.ar(Saw.ar(----,----),-----,0.2);
        ---=EnvGen.kr(Env.perc(0,----),doneAction:2);
        Out.ar(----,----*----)
        }).----(s)
)
```

Crea tus propios SynthDef.

Clase 5: UGens como argumentos de UGens

Como hemos visto los UGens tienen argumentos específicos que pueden ser determinados.

Pulse.ar (frecuencia, ancho del pulso, mul, adición)

A estos argumentos se les pueden asignar valores fijos:

```
s.boot;

{Pulse.ar(456,0.34,0.45,0)}.play;
```

También podemos meter a los UGens dentro de una estructura de SynthDef y cambiar sus argumentos desde afuera:

```
(
SynthDef(\cacahuate, {|gate=1, freq=456, ancho=0.34, amp=0.45|
                        var sig, env;
                        sig=Pulse.ar(freq,ancho,amp,0);
                        env=EnvGen.kr(Env.asr(2,1,1),gate, doneAction:2);
                        Out.ar(0,sig*env);
                        }).send(s)
)

c=Synth(\cacahuate)
c.set(\freq, [1,2,3,4,5,6].choose*456)
c.set(\ancho, [1,2,3,4,5,6].choose*0.1)
c.set(\gate, 0)
```

Ahora veremos una manera diferente de hacer que los argumentos de un UGen cambien de valor. Primero que nada observemos los datos que arroja un UGen en particular, digamos un

SinOsc.

{SinOsc.ar(1,0,1,0)}.play // no se preocupen, no se oye nada por que la frecuencia es muy baja !

Este UGen nos da un ciclo por segundo de una onda senoidal. La amplitud es de 1, así que el recorrido de la onda parte de 0, sube a 1, baja a -1 y regresa a 0 en un segundo. Podemos decir que el rango de esta función va de -1 a 1. El tercer elemento del SinOsc es el mul y el valor que asignemos ahí multiplicará a los valores de nuestra función. Siempre pedimos que sea 1 el máximo valor por que siempre hemos usado el mul como valor para la amplitud de un sonido. Lo que sucede con el rango de nuestra senoide al ser multiplicado por distintos valores del mul podemos observarlo en las siguientes líneas de código:

[-1,1] * 1
[-1,1] * 0.5
[-1,1] * 0.25
[-1,1] * 0
El cuarto argumento del SinOsc es el add o adición. Siempre hemos usado el add en cero por default. Este valor se suma a la función de nuestro UGen y se refleja en el rango. En el ejemplo que estamos viendo nuestro rango es de [-1, 1] por eso al sumarle 0 no se ve afectado.

[-1,1] + 0

Ahora veamos cómo se afecta este rango al combinar diferentes valores del mul y el add.

[-1,1] * 1 + 1
[-1,1] * 10 + 10
[-1,1] * 100 + 100
[-1,1] * 100 + 200

Si nos fijamos en el ejemplo [-1,1]*100+200 vemos que nuestro rango va de 100 a 300. Entonces sabemos ahora que un SinOsc con frecuencia 1, fase 0, mul 100 y add 200 nos da una onda sinoidal que varía su frecuencia entre 100 y 200.

```
{SinOsc.ar(1,0,100,200)}.play
```

Claro que cuando lo queremos oír no podemos por que la frecuencia sigue siendo de 1 y es muy baja. Pero podemos aprovechar que el juego entre el mul y el add nos arroja números que pueden ser utilizados como valores para la frecuencia de OTRO SinOsc. Así que simplemente colocamos el código anterior en el argumento de frecuencia de otro SinOsc:

```
{SinOsc.ar(SinOsc.ar(1,0,100,200))}.play
```

Ahora tenemos una senoide que cambia su argumento de frecuencia moviendo su valor entre 100 y 200 una vez por segundo. Este ejemplo en particular es llamado FM o Frecuencia Modulada por que estamos modulando o cambiando la frecuencia de un sonido periódicamente. Veamos como se escucha con distintos valores de freq, mul y add. Vamos a cambiar el *audio rate* por *kontrol rate* en el SinOsc que esté como argumento de frecuencia por que este SinOsc no esta generando una señal sino que solo controla un argumento.

```
{SinOsc.ar(SinOsc.kr(10,0,100,200))}.play
{SinOsc.ar(SinOsc.kr(1,0,100,1000))}.play
{SinOsc.ar(SinOsc.kr(10,0,100,1000))}.play
{SinOsc.ar(SinOsc.kr(100,0,10000,10200))}.play
```

Pero bueno, mejor usemos un SynthDef.

```
(
SynthDef(\fm, {|gate=1, freq=1, mul=100, add=1000, amp=1|
        var modulacion, sig, env;
        modulacion=SinOsc.kr(freq,0,mul,add);
        sig=SinOsc.ar(modulacion,0,amp);
```

```
        env=EnvGen.kr(Env.asr(2,1,2),gate,doneAction:2);
        Out.ar(0,sig*env);
        }).send(s)
)

m=Synth(\fm)
m.set(\freq, 10, \mul, 500, \add, 1000)
m.set(\gate, 0)
```

Podemos usar cualquier oscilador como controlador de otro UGen y de igual manera que hemos controlado la frecuencia de un SinOsc podemos controlar su amplitud o cualquier otro argumento. Cuando modulamos la amplitud de un oscilador se le llama AM o Amplitud Modulada. Veamos como suena colocando un SinOsc.kr en el argumento de amplitud de un SinOsc.ar. Cuidado con los valores que escogemos para el mul y el add ya que al estar trabajando con la amplitud no debemos superar el 1.

```
{SinOsc.ar(440,0,SinOsc.kr(1,0,0.5,0.5))}.play
```

Escogí los valores 0.5 para el mul y 0.5 para el add por que me arrojan un rango entre 0 y 1 que son el máximo y el mínimo recomendados para la amplitud.

```
[-1,1]*0.5+0.5
```

Hagamos un SynthDef para generar una síntesis AM.

```
(
SynthDef(\am, {|gate=1, portadora=1000, moduladora=10, amp=1|
        var modulacion, sig, env;
        modulacion=SinOsc.kr(moduladora,0,0.5,0.5);
        sig=SinOsc.ar(portadora,0,modulacion);
        env=EnvGen.kr(Env.asr(2,1,2),gate,doneAction:2);
        Out.ar(0,sig*env*amp);
```

```
        }).send(s)
)

a=Synth(\am)
a.set(\moduladora, 2000)
a.set(\gate, 0)
```

La FM y la AM son formas de síntesis que poseen características muy interesantes que veremos en un curso posterior.

LFNoise0, LFNoise1 y LFNoise2

Veamos ahora un nuevo generador de ruido que podemos usar como UGen de control como en los ejemplos anteriores. El LFNoise o Ruido de Baja Frecuencia tiene 3 modalidades.

LFNoise0: genera valores aleatorios saltando de uno a otro valor.
LFNoise1: genera valores aleatorios creando una línea que recorre los valores intermedios entre estos.

LFNoise2: genera valores aleatorios creando una curva cuadrática que recorre los valores intermedios entre estos. Esto es muy útil cuando queremos controlar frecuencias ya que las frecuencias altas ocupan la mayoría del rango de frecuencias y debemos de compensar esto para tener igual oportunidad de obtener las frecuencias bajas.

Escuchemos cómo se oyen estos generadores de ruido por sí mismos. Los argumentos de los tres generadores son los mismos : freq, mul y add.

```
s.quit;
s=Server.internal.boot

{LFNoise0.ar(1000,1,0)}.scope
```

```
{LFNoise1.ar(1000,1,0)}.scope
{LFNoise2.ar(1000,1,0)}.scope
```

Observando el osciloscopio podemos ver gráficamente la característica de cada tipo de LFNoise. Ahora usémoslos como controladores. El principio del rango es igual que en el ejemplo del SinOsc por que, como la mayoría de los osciladores, los LFNoise trabajan con un rango inicial de -1 a 1.

```
[-1,1]*9990+10010 // números entre 20 y 20000
[-1,1]*50+350 // números entre 20 y 20000

{SinOsc.ar(LFNoise0.kr(10,9990,10010),0,0.5)}.scope // Cambia su frecuencia por salto entre 20hz y 20000hz 10 veces por segundo.
{SinOsc.ar(LFNoise1.kr(10,9990,10010),0,0.5)}.scope // Cambia su frecuencia linealmente entre 20hz y 20000hz 10 veces por segundo.
{SinOsc.ar(LFNoise2.kr(10,9990,10010),0,0.5)}.scope // Cambia su frecuencia cuadráticamente entre 20hz y 20000hz 10 veces por segundo.
```

Para obtener el valor del mul y el add a partir de un rango conocido podemos utilizar la siguiente fórmula:

```
mul=(máximo-mínimo)/2
add=mul+mínimo
```

Por ejemplo, para obtener un rango entre 20hz y 20000hz

```
mínimo=20
máximo=20000
mul=(máximo-mínimo)/2=(20000-20)/2=9990
add=mul+mínimo=9990+20=10010

[-1,1]*9990+10010
```

Así, con esta fórmula podemos hacer un SynthDef creando argumentos para el mínimo y el máximo del rango en el control de una frecuencia:

```
s.quit;
s=Server.local.boot;

(
SynthDef(\controlNoise, {|gate=1, freq=10, min=20, max=20000, amp=1|
		var control, mul, add, sig, env;
		mul=(max-min)/2;
		add=mul-min;
		control=LFNoise2.kr(freq,mul,add);
		sig=SinOsc.ar(control,0,amp);
		env=EnvGen.kr(Env.asr(2,1,2),gate,doneAction:2);
		Out.ar(0,sig*env);
		}).send(s)
)

n=Synth(\controlNoise)
n.set(\freq, 10, \min, 500, \max, 1000)
n.set(\gate, 0)
```

Clase 6: Tdef

Tdef

.do

Necesitamos conocer una herramienta que nos sirve como núcleo de nuestro Tdef. Estamos hablando del n.do. Veamos un ejemplo:

```
"gggggg9g9g9g".postln

2.do{ "gggggggggggggggggg9999ggg999ggg999gg".postln}
```

Intuitivamente podemos pensar que la línea anterior escribe 2 veces el string "gggggggggggggggggg9999ggg999ggg999gg". En realidad así es. Entonces definamos .do como un método que nos hace n veces lo que le pongamos dentro de las llaves { }. Para hacer más interesante la cosa agregemos el mensaje .scramble que desordena una cadena de caractéres o los elementos de un array. Mira:

```
[0,1,2,3,4,5];
[0,1,2,3,4,5].scramble

"gggggggggggggggggg9999ggg999ggg999gg".postln
"gggggggggggggggggg9999ggg999ggg999gg".scramble.postln
```

Ahora con el .do

```
50.do{"gggggggggggggggggg9999ggg999ggg999gg".scramble.postln}

50.do{ [0,1,2,3,4,5].scramble.postln}
```

Podemos poner más de una acción dentro de la función. Solo hay que escribir punto y coma ; entre cada acción distinta.

```
50.do{
        "gggggggggggggggggg9999ggg999ggg999gg".scramble.postln;
        [0,1,2,3,4,5].scramble.postln;
        [1,2,3,4,5,6].choose.postln;
        }
```

Podemos poner cualquier cosa dentro de las llaves del .do. Pongamos un oscilador:

```
{SinOsc.ar(rrand(300,5000),0,1/10)}.play

10.do{{SinOsc.ar(rrand(300,5000),0,1/10)}.play}
```

O podemos llamar a un Synth. Primero hagamos el SynthDef y luego lo llamamos dentro de un .do.

```
SynthDef(\hacer, {|freq=400|
Out.ar(0,SinOsc.ar(freq,0,1/10)*EnvGen.kr(Env.perc(0.01,5),doneAction:2))}).send(s)

10.do{Synth(\hacer, [\freq, rrand(300,5000)])}
```

Otro ejemplo de cómo usar el .do lo tenemos dentro de un SynthDef con reverb:

```
(
SynthDef(\rev, {|gate=1|
        var sen, env;
        sen=Impulse.ar(1);
        5.do{sen=AllpassN.ar(sen, 0.1, rrand(0.03,0.04), 1.5)};
        env=EnvGen.kr(Env.asr(0,1,0),gate, doneAction:2);
        Out.ar(0,sen*env);
        }).send(s);
)

r=Synth(\rev)
r.set(\gate, 0)
```

Tdef

¿Qué es?, ¿Para qué sirve?

1- Tdef es el nombre que se le da a la estructura que nos permite generar loops o repeticiones.
2- Es de gran utilidad para la creación de patrones rítmicos o para modificar, de manera automática, los argumentos de nuestros Synths.
3- Los Tdef funcionan por medio de un reloj al que podemos cambiar su velocidad.
4- Los Tdef pueden modificarse sin necesidad de ser apagados.

Como vimos en los ejemplos anteriores el .do realiza n veces una acción que está dentro de las llaves de función { } y lo hace inmediatamente. Pero, ¿Qué pasa si queremos que lo haga con una pausa entre cada vez? Para eso necesitamos recurrir a otra de las estructuras o *templates* que SuperCollider nos brinda: Tdef.

Tdef viene de las palabras *Task definition* y es similar en su construcción al SynthDef. Veamos como se construye y para que sirve cada una de sus partes.

```
(
Tdef(\x, { // Escribimos Tdef, abrimos paréntesis y ponemos un string con el nombre con el que queremos identificar el Tdef. Ponemos coma y abrimos una llave que englobará todo el contenido de nuestro Tdef.
        50.do{      // Después ponemos un .do como hemos hecho anteriormente.
        [0,1,2,3,4].scramble.postln; //ponemos una acción que sera ejecutada 50 veces
        0.05.wait; // Añadimos otra acción mas que será un número con el mensaje .wait, este número representa el tiempo en segundos que esperará antes de volver a hacer la acción.
                }
    }); // Cerramos la llave englobadora del Tdef y el paréntesis.
    )
```

Después para hechar a andar el Tdef lo hacemos refiriéndonos al Tdef por su nombre con el mensaje .play como se muestra a continuación:

```
Tdef(\x).play
Tdef(\x).stop
```

Para detener el Tdef usamos el mensaje .stop. Si queremos que el Tdef tenga un .do con un número infinito de repeticiones utilizamos la palabra inf en vez de un número finito.

```
(
Tdef(\x, {
		inf.do{
			"ggggg_____ggggsupercollider rules!
gggggg****gg9999ggg999ggg999gg".scramble.postln;
			0.15.wait;
			}
		});
		)
Tdef(\x).play
Tdef(\x).stop
```

Es MUY IMPORTANTE que cuando usemos inf.do nos aseguremos de escribir un .wait dentro por que si no le estaremos pidiendo a SuperCollider que haga un número infinito de acciones en una cantidad infinitamente pequeña de tiempo.

Un sinónimo de inf.do es loop.

```
(
Tdef(\x, {
		loop{
			"ggggggggggggggggg9999ggg999ggg999gg".scramble.postln;
```

```
                0.05.yield;
                }
        });
        )
```

un sinónimo de .wait es .yield

```
Tdef(\x).play
Tdef(\x).stop
```

Dentro de un .do podemos poner otros .do en lo que llamamos un arreglo anidado.

```
(
1.do{
        4.do{ "---------////----------------------".postln;};
        10.do{"======;;;;;=======".postln;};
        5.do{"*******////########".postln;}
        }
)
```

Observemos cómo se comporta el arreglo anidado dentro de un Tdef con inf.do. Nótese que cada .do tiene su propio .wait.

```
(
Tdef(\x, {
        inf.do {
                4.do {
                "---------////----------------------".scramble.postln;
                0.25.wait;
                };
                10.do {
```

```
            "======;;;;;=======".scramble.postln;
            0.05.wait;
            };
            5.do{
            "*******////########".scramble.postln;
            0.15.wait;
            }
            }
        });
)

Tdef(\x).play
Tdef(\x).stop
```

Ahora viene lo bueno. Podemos llamar a un Synth dentro de un Tdef para que no tengamos que estar picandole como locos con el *mouse*. Mira:

```
(
SynthDef(\uno,{|out=0|
            var sen, env;
            sen=SinOsc.ar(820,0,0.8)!2;
            env=EnvGen.kr(Env.perc(0,0.1),doneAction:2);
            Out.ar(out,sen*env)
            }).send(s)
)

Synth(\uno)
```

Este Tdef dispara el Synth(\uno) dos veces por segundo o cada 0.5 segundos.

```
(
Tdef(\x,{
	inf.do{
			Synth(\uno);
		0.5.wait;
		}
	})
)

Tdef(\x).play
Tdef(\x).stop
```

Probablemente se habrán dado cuenta de que al darle play al Tdef comienza a funcionar con una pequeña latencia. Para evitar esto escribimos el mensaje .quant . Ejemplo:

```
Tdef(\x).quant_(0).play
Tdef(\x).stop
```

Como pueden ver el mensaje .quant lleva escrito un argumento que en este caso es 0 indicando que no haya latencia al comenzar el Tdef. Notar el guión bajo antes de los paréntesis para el argumento.

Otro ejemplo sencillo incorporando un XLine dentro del Synth. El XLine hace una línea de números en base a tres argumentos:

XLine.kr(punto de partida, punto final, tiempo que se tarda en ir de un punto a otro)

```
(
SynthDef(\cuatro2,{|out=0,gate=1,frecuencia=100,mul=0.6|
		var sen, env;
		sen=Saw.ar(XLine.kr(12200,120,0.13),mul)!2;
```

```
        env=EnvGen.kr(Env.perc(0,0.15),gate,doneAction:2);
        Out.ar(out,sen*env)
        ).send(s)
        )

Synth(\cuatro2)

(
Tdef(\gh,{
    inf.do{
        Synth(\cuatro2);
        0.58.wait;
        }}
        )
        )

Tdef(\gh).play
Tdef(\gh).stop
```

Hagamos otro Synth con la capacidad de cambiar su frecuencia desde afuera con un argumento "frecuencia".

```
(
SynthDef(\dos,{|out=0,frecuencia=100|
        var sen, env;
        sen=SinOsc.ar(frecuencia,0,0.8)!2;
        env=EnvGen.kr(Env.perc(0,0.1),doneAction:2);
        Out.ar(out,sen*env)
        ).send(s)
)
```

Este Tdef es como un LFNoise0.kr modulando a un SinOsc.ar cambiando su frecuencia cada

0.18 segundos.

```
(
Tdef(\x1,{
	inf.do{
		Synth(\dos,[\frecuencia,rrand(100,4000)]);
		0.18.wait
	}
	})
)

Tdef(\x1).play
Tdef(\x1).stop
```

Ahora fíjense como se comporta un Tdef con .do anidados disparando Synths relacionados con *strings* que se escriben en la *post window*.

```
(
SynthDef(\tres,{|out=0,frecuencia=100,mul=0.8|
		var sen, env;
		sen=SinOsc.ar(frecuencia,0,mul);
		env=EnvGen.kr(Env.perc(0,0.1),doneAction:2);
		Out.ar(out,sen*env)
		).send(s)
)

(
Tdef(\x2,{
		inf.do{
			3.do{
```

```
                        Synth(\tres,[\frecuencia,rrand(200,300),\mul,rrand(0,0.8)]);
                "--------////----------".scramble.postln;
        0.1.wait;
                };
                3.do{
                        Synth(\tres,[\frecuencia,1300,\mul,rrand(0,0.8)]);
        "======;;;;;=======".scramble.postln;
                0.07.wait;
                };
                5.do{
                Synth(\tres,[\frecuencia,rrand(2700,3500),\mul,rrand(0,0.8)]);
        "***////##".scramble.postln;
                0.17.wait;
                }
        }
    })
)

Tdef(\x2).play
Tdef(\x2).stop

Lo mismo pero solo los Synth

(
SynthDef(\cuatro,{|out=0,gate=1,frecuencia=100,mul=0.8|
        var sen, env;
        sen=Saw.ar(frecuencia,mul)!2;
        env=EnvGen.kr(Env.perc(0.06,0.15),gate,doneAction:2);
        Out.ar(out,sen*env)
        ).send(s)
)
```

```
(
Tdef(\x2,{
        inf.do{
                3.do{
                        Synth(\cuatro,[\frecuencia,rrand(200,300),\mul,rrand(0,0.4)]);
            0.1.wait;
                        };
        2.do{
            Synth(\cuatro,[\frecuencia,130,\mul,rrand(0,0.4)]);
            .06.wait;
            };
            4.do{
            Synth(\cuatro,[\frecuencia,rrand(270,350),\mul,rrand(0,0.4)]);
        0.17.wait;
        }
        }
    })
)

Tdef(\x2).play
Tdef(\x2).stop

(
Tdef(\x2,{
        inf.do{
        3.do{
            Synth(\cuatro,[\frecuencia,4000,\mul,rrand(0,0.4)]);
            0.071.wait;
            };
            8.do{
```

```
                Synth(\cuatro,[\frecuencia,1300,\mul,rrand(0,0.4)]);
                0.057.wait;
                };
        4.do{
                Synth(\cuatro,[\frecuencia,345,\mul,rrand(0,0.4)]);
        0.17.wait;
                };
        14.do{
        Synth(\cuatro,[\frecuencia,45,\mul,rrand(0,0.4)]);
                0.07.wait;
                }
                }
        })
)

Tdef(\x2).play
Tdef(\x2).stop
```

Hasta ahora hemos trabajado con Tdefs que disparan y controlan los argumentos de Synths con envolventes percusivas. Esto implica que nuestros Tdefs tienen que disparar los Synths. Si queremos usar un Tdef para controlar un Synth con envolvente del tipo ASR entonces tenemos que disparar el Synth solo una vez al principio y luego el Tdef modificará los argumentos de nuestro Synth mientras este prendido. Para detenerlo hay que apagar el Tdef y luego el SynthDef.

```
(
SynthDef(\cinco,{|out=0,frecuencia=100,cutoff,amp=0.3,gate=1|
        var sen,env;
                sen=Saw.ar(frecuencia,amp)!2;
                env=EnvGen.kr(Env.asr(0.3,1,2),gate,doneAction:2);
                Out.ar(out,sen*env)
```

```
        ).send(s)
        )
a=Synth(\cinco)
a.set(\gate,0)
```

Fijense como podemos prender el Synth desde el Tdef si lo colocamos antes del .do

```
(
Tdef(\x3, {  a=Synth(\cinco);
        inf.do{
                a.set(\frecuencia,rrand(1000,120));
                0.3.wait
        }
    })
)

Tdef(\x3).play
Tdef(\x3).stop;a.set(\gate,0)
a.set(\gate,0) // Hay que apagar el Synth despues del Tdef si ya no queramos usarlo.

(
SynthDef(\cinco,{|out=0,frecuencia=100,cutoff=300,amp=0.3,gate=1|
        var sen,env,filtro;
        sen=Saw.ar(frecuencia,amp)!2;
        filtro=RLPF.ar(sen,cutoff,0.9);
        env=EnvGen.kr(Env.asr(0.3,1,2),gate,doneAction:2);
        Out.ar(out,filtro*env)
    }).send(s)
)

d=Synth(\cinco)
```

```
d.set(\gate,0)
```

En este ejemplo prendemos el Synth desde afuera y antes de prender el Tdef.

```
a=Synth(\cinco);

(
Tdef(\x3,{
        inf.do{
                a.set(\frecuencia,rrand(100,220),\cutoff,8000);
                0.1.wait
        }
    }
    )
)

Tdef(\x3).play,
Tdef(\x3).stop;
a.set(\gate,0)
```

Observen que pasa si prendemos un Tdef que modifica los argumentos de un Synth que aún no ha sido prendido.

```
(
Tdef(\x3,{
        inf.do{
                a.set(\frecuencia,rrand(200,420),\cutoff,
[12000,8400,9600,7500,3500,6700,6000].choose);
                0.1.wait;
        }
    }
```

```
        )
)

Tdef(\x3).play
```

Nos dice que no encuentra el Synth : FAILURE /n_set Node not found.

Prendemos el Synth y la falla desaparece.

```
a=Synth(\cinco);
Tdef(\x3).stop;
a.set(\gate,0)
```

A continuación se muestran varios ejemplos de la mancuerna Tdef - SynthDef

```
(
Tdef(\x3,{
        inf.do{a.set(\frecuencia,[60,62,64,65,67,69,71].choose,\cutoff,
[200,400,600,1400,1500,2700,3000].choose);0.1.wait
        }}
        )
)

a=Synth(\cinco);Tdef(\x3).play
Tdef(\x3).stop;a.set(\gate,0)

(
Tdef(\x3,{
        inf.do{
        3.do{a.set(\frecuencia,[60.3453,62.204,64.32333,65.5324,67.209,69.42,71].choose,
\cutoff, [2300,2400,600,3400,1500,2700,3000].choose);0.21.wait};
```

```
	4.do{a.set(\frecuencia,[120,632,614,65,67,269,741].choose, \cutoff
[14200,4040,6500,8400,5500,6700,3000].choose);0.1.wait};
	12.do{a.set(\frecuencia,[610,622,64,635,367,69,71].choose,\cutoff,
[2200,4400,6600,14040,1500,2700,3000].choose);0.071.wait};
	3.do{a.set(\frecuencia,[640,62,64,665,67,639,71].choose,\cutoff,
[12200,400,600,1400,15300,2700,3000].choose);0.11.wait}
	}}
	)
	)

a=Synth(\cinco);
Tdef(\x3).play
Tdef(\x3).stop;
a.set(\gate,0)

(
SynthDef(\seis,{|out=0,frecuencia=60,cutoff=300,amp=0.21,gate=1|
		var sen,env,filtro;
		sen=Saw.ar(frecuencia,amp)!2;
		filtro=RLPF.ar(sen,MouseX.kr(200,15000),0.9);
		env=EnvGen.kr(Env.asr(0.3,1,2),gate,doneAction:2);
		Out.ar(out,filtro*env)
		}).send(s)
)

b=Synth(\seis);
b.set(\gate,0)
```

Aquí vemos como el número que antecede al .do puede ser aleatorio y se define de modo diferente en cada vuelta. Utilizamos un +1 en cada .do para evitar el 0.

```
(
Tdef(\x4,{
	inf.do{
	1+(10.rand).do{b.set(\frecuencia,[60,65].choose);0.19.wait};
	1+(5.rand).do{b.set(\frecuencia,[66,67].choose*4);0.19.wait};
	1+(7.rand).do{b.set(\frecuencia,[61,65].choose*16);0.19.wait}
	}}
	)
)

Tdef(\x4).play
Tdef(\x4).stop;
b.set(\gate,0)
```

Ahora veamos como podemos construir una batería acompañando a un instrumento melódico por medio de Tdefs. Primero hacemos nuestros SynthDefs para cada elemento de la batería como bombo, tarola y hi hat y también el SynthDef para nuestro instrumento melódico que aquí tiene el revelador nombre de "seis".

```
(
(
SynthDef(\bombo,{|out=0,gate=1,frecuencia=60|
		var sen, env;
		sen=Pan2.ar(SinOsc.ar(frecuencia,pi*0.25,0.8),0,0.9);
		env=EnvGen.kr(Env.perc(0,0.13),gate,doneAction:2);
		Out.ar(out,sen*env)
		}).send(s)
		);

(
```

```
SynthDef(\tarola,{|out=0,gate=1,frecuencia=660|
		var sen, env,ruido;
		ruido=LPF.ar(WhiteNoise.ar(0.34),XLine.kr(12000,200,0.2),0.9);
		sen=Pan2.ar(SinOsc.ar(frecuencia,0,0.114)+ruido,0,0.8);
		env=EnvGen.kr(Env.perc(0,0.153),gate,doneAction:2);
		Out.ar(out,sen*env)
		}).send(s)
);

(
SynthDef(\hi,{|out=0,gate=1|
		var sen, env;
		sen=HPF.ar(WhiteNoise.ar(0.5),2000,0.9);
		env=EnvGen.kr(Env.perc(0,0.13),gate,doneAction:2);
		Out.ar(out,Pan2.ar(sen*env,0,0.5))
		}).send(s)
	)
)
(
SynthDef(\seis,{|out=0,frecuencia=60,cutoff=300,amp=0.21,gate=1|
		var sen,env,filtro;
		sen=Saw.ar(frecuencia,amp);
		filtro=RLPF.ar(sen,MouseX.kr(200,15000),0.9);
		env=EnvGen.kr(Env.asr(0.3,1,2),gate,doneAction:2);
		Out.ar(out,filtro*env)
		}).send(s)
)
```

Como ven los Synths de la batería tienen una envolvente percusiva y el de la melodía tiene envolvente ASR.

Prendemos el Synth "seis".

Vamonos !!!

```
(
b=Synth(\seis);
(
Tdef(\bat,{
	inf.do{
	2.do{Synth(\bombo);0.19.wait};
	1.do{Synth(\tarola);(0.19*2).wait}
	}}
)
);
Tdef(\bat).play;
(
Tdef(\hihat,{
	inf.do{Synth(\hi);0.19.wait
	}}
	)
);

Tdef(\hihat).play;
(
Tdef(\x4,{
	inf.do{
	1+(10.rand).do{b.set(\frecuencia,[60,65].choose);0.19.wait};
	1+(5.rand).do{b.set(\frecuencia,[66,67].choose);0.19.wait};
	1+(7.rand).do{b.set(\frecuencia,[61,65].choose);0.19.wait}
	}}
	)
```

```
        );
Tdef(\x4).play
)

// Bueno, ya.

Tdef(\bat).stop;Tdef(\hihat).stop;Tdef(\x4).stop;b.set(\gate,0)
```

Aparte de todo para grabar con un bitRate de 16 escriben

```
s.recSampleFormat="int16"
```

Clase 7: If switches y relojes

If

Con el mensaje if podemos condicionar la ejecución de funciones. Este mensaje recibe una expresión que al ser evaluada debe regresar los valores *true* o *false* (valores booleanos). Si la expresión es cierta, se ejecutará la función que se encuentra en el segundo argumento, si es falsa, se ejecutará la función que se encuentra en el tercer argumento.

El mensaje *if* nos regresará el resultado de la función que haya sido evaluada. Si no hay ninguna función para ser evaluada en el caso de que la expresión haya sido falsa, el resultado del mensaje será nil.

```
if ( expression, { trueFunction }, { falseFunction } )
```

```
(
if(
	2.0.rand.postln > 1,		// la expresión booleana

	{"la expresion fue cierta" },	// la función a evaluar si la expresión es cierta

	{"la expresion fue falsa"};	// la función a evaluar si la expresión es falsa
)
)

// si la expresión es falsa y no hay ninguna función que sea evaluada, if nos dará nil

if ( 1000.rand > 900, {"mayor que 900"})
// Evalúa el siguiente código varias veces

(
```

```
{
var opciones;

opciones  =     [ \aditiva, \substractiva ];

if ( opciones.choose == \aditiva,
        {"aditiva".postln;
         a = Mix.fill(10, {SinOsc.ar(exprand(30.0, 300.0), 0, 0.07)}) },
        {"substractiva".postln;
         a = Mix.fill(10, {BPF.ar(BrownNoise.ar, f = exprand(300.0, 3000.0), 0.5/f, 4)})}
        );

a ! 2 }.scope

)

// Evalúa las siguientes líneas
```

Expresiones booleanas

```
1 < 2       // menor que

1 > 2       // mayor que

1 == 2      // igual que (atención: es muy común cometer el error de sólo poner un signo de igual ( = ) en vez de dos ( == )

1 != 2      // distinto que

1 <= 2      // menor o igual que

1 >= 2      // mayor o igual que
```

```
1.odd       // ¿es un número non?

1.even      // ¿es un número par?
```

Evaluando algunas funciones

Si quisieramos evaluar una función es necesario seguir algunos pasos. Primero construimos la función que queremos evaluar. Por ejemplo:

```
x={|n| if(n>10,{"es mayor a 10"},{"es menor a 10"})} //construimos la funcion x en donde n (argumento) es nuestra incognita; compilamos el codigo y Supercollider nos contesta 'a Function'
```

Para darle valores a la función x es necesario la siguiente línea de código:

```
x.value(1) // en donde dentro de los paréntesis asignamos el valor que tomara n
```

Supongamos que queremos evaluar lo siguiente:

```
a={|n| if(n>10,{"es mayor a 10"},{"es menor a 10"})}
a.value(45)

a={|n| if(n.even,{"es numero par"},{"es numero impar"})}
a.value(8)

a={|n| if(n!=10,{"es distinto a 10"},{"es 10"})}
a.value(10)

//Ahora con Tdef
(
a={|n| if(n>10,{"es mayor a 10"},{"es menor a 10"})};
Tdef(\hu,{
```

```
        inf.do{a.value(rrand(2,20)).postln;
                0.5.wait
                }}
                )
)
Tdef(\hu).play
Tdef(\hu).stop
```

RELOJES

iteración

Una operación presenta valores de entrada y de salida. En el siguiente ejemplo a y b toman los valores de entrada y salida

a = 2

b = a + 1

Una iteración es un proceso en el que el valor de salida de una operación se utiliza como un nuevo valor de entrada. Ejemplo

```
a=0
b=a+1
a=b
b=a+1
a=b
b=a+1
a=b
b=a+1
a=b
```

b=a+1
a=b
b=a+1
a=b
b=a+1

Pero este algoritmo se simplifica de esta manera.

b=0

b=b + 1

Si no damos la información previa, éste algoritmo podría interpretarse como una contradicción por lo cual es importante que seamos concientes de lo que implica el algoritmo de iteración dentro del lenguaje de SuperCollider. Matemáticamente el algorítmo simplificado significa esto:

b = 0

b = b + 1 => 0 = 0 + 1 ! Contradicción

Pero dentro del lenguaje de SuperCollider y otros varios este algorítmo significa la iteración que hemos explicado.

Mediante una simple interación como la que vimos, podemos construir una estructura que será muy útil.

Esta estructura simula un contador que nos devuelve una secuencia de números extraída del algoritmo x = x + 1
Veamos como funciona dentro de un Tdef:

```
(
Tdef(\reloj,{var contador = 0;
inf.do{contador = contador + 1;
        contador.postln;
        0.1.wait
        }}
        )
)

Tdef(\reloj).play
Tdef(\reloj).stop
```

Pero esta estructura nos devuelve una lista de números ¡¡¡ infinitos !!!
En ciertas circunstancias necesitaremos limitar la secuencia a cierto número máximo después del cual se reenicia la cuenta.
Para esto es necesario utilizar el modulo (%)

```
(
Tdef(\reloj,{
        var contador = 0;

        inf.do{contador = contador + 1;
        (contador % 10).postln;
        0.5.wait
}}
)
);

Tdef(\reloj).play
Tdef(\reloj).stop
```

Switch

Es una herramienta que nos permite evaluar una variable y, dependiendo de su valor, asignar acciones en respuesta. Observen como en el ejemplo siguiente podemos asignar un valor a la variable 'a' y el switch realizará las acciones que le indiquemos dependiendo si 'a' vale 0, 1, 2, 3 ó 4.

```
(

a = 3;

switch(a, 0,{'a es igual a cero'.postln;},
        1,{'a es igual a uno'.postln},
        2,{'a es igual a dos'.postln},
        3,{'a es igual a tres'.postln},
        4,{'a es igual a cuatro'.postln}
    );
)
```

Podemos asignar más de una acción dentro de la función de respuesta. Llamemos a un Synth además del postln.

```
(
SynthDef(\switch, {|freq=100|
        var sig, env;
        sig=Pulse.ar(freq,Line.kr(0.5,0.01,0.1))!2;
        env=EnvGen.kr(Env.perc(0,01,0.1),doneAction:2);
        Out.ar(0,sig*env);
        }).send(s);
);
```

```
Synth(\switch)

(

a=1;
switch(a, 0,{'a es igual a cero'.postln; Synth(\switch, [\freq, 100])},
        1,{'a es igual a uno'.postln; Synth(\switch, [\freq, 2000])},
        2,{'a es igual a dos'.postln; Synth(\switch, [\freq, 300])},
        3,{'a es igual a tres'.postln; Synth(\switch, [\freq, 400])},
        4,{'a es igual a cuatro'.postln; Synth(\switch, [\freq, 500])},
        24,{'a es igual a cuatro'.postln; Synth(\switch, [\freq, 500])}
    );
)
```

En definitiva el *switch* se puede ver como un conjunto de varios *if* pero que nos ahorra trabajo de escritura. Vean cómo resolvemos el código anterior pero hecho con *if*.

```
(
a=4;
if(a==0,{'a es igual a cero'.postln; Synth(\switch, [\freq, 100])});
if(a==1,{'a es igual a uno'.postln; Synth(\switch, [\freq, 200])});
if(a==2,{'a es igual a dos'.postln; Synth(\switch, [\freq, 300])});
if(a==3,{'a es igual a tres'.postln; Synth(\switch, [\freq, 400])});
if(a==4,{'a es igual a cuatro'.postln; Synth(\switch, [\freq, 500])});
)

(
(
SynthDef(\puls, {|freq|
        var sig, env;
        sig=Pulse.ar(300,Line.kr(0.5,0.01,0.1));
```

```
            env=EnvGen.kr(Env.perc(0,01,0.1),doneAction:2);
            Out.ar(0,sig*env);
            ).send(s);
);

(
SynthDef(\sin, {|freq|
            var sig, env;
            sig=SinOsc.ar(2000,0,0.3);
            env=EnvGen.kr(Env.perc(0,01,0.1),doneAction:2);
            Out.ar(0,sig*env);
            }).send(s);
);

(
SynthDef(\saw, {|freq|
            var sig, env;
            sig=Saw.ar(2000,0.8);
            env=EnvGen.kr(Env.perc(0,01,0.1),doneAction:2);
            Out.ar(0,sig*env);
            }).send(s);
);
)

(
Tdef(\sw,{
            var contador = 0;
            inf.do{contador = contador + 1;
            (contador %3).postln;
                        switch(contador%3, 0,{Synth(\puls)},
                                                1,{Synth(\sin)},
```

```
                                                    2,{Synth(\saw)}
                                            );
                        0.3.wait
                        }}
                )
        )

Tdef(\sw).play
Tdef(\sw).stop

(
SynthDef(\saw, {|freq|
        var sig, env;
        sig=Saw.ar(freq,0.8)!2;
        env=EnvGen.kr(Env.perc(0,01,0.1),doneAction:2);
        Out.ar(0,sig*env);
        }).send(s);
);

(
Tdef(\sw,{var contador=0;
    inf.do{contador=contador+1;
    (contador%3).postln;
    Synth(\saw,[\freq,((contador%[3,10].choose)+1)*100]);
                        0.1.wait
                        }}
                )
        )

Tdef(\sw).play
Tdef(\sw).stop
```

Clase 8: Buffer

Basada en una clase de Sergio Luque

```
s = Server.internal; //prendemos el Servidor de esta manera para correr los ejemplos de la clase 8
s.boot;
```

buffer

Los Buffers son arrays de números de punto flotante (con decimales) con un pequeño encabezado descriptivo.
Son colocados en un array global y están indexados por números enteros, empezando por el cero.
Los vamos a utilizar para accesar o escribir: archivos de audio, tablas, líneas de delay, envolventes o para cualquier otra cosa que pueda utilizar un array de números de punto flotante.

Solo vamos a ver una de las dos formas de cargar un archivo de audio:

1- Object style con Buffer.read

Buffer.read asigna un buffer e inmediatamente lee un archivo de audio en él.

```
Buffer.read (server, path, startFrame = 0, numFrames = -1, completionMessage);
```

server: el servidor en donde crearemos el buffer. Generalmente pondremos: s
path: la dirección del archivo de audio.
startFrame: a partir de qué cuadro del archivo vamos a leer. 0 si queremos leerlo desde el principio (éste es el default).
numFrames: número de cuadros que vamos a leer. -1 si queremos que lea todo el archivo. (éste es el default).

completionMessage: una función que será evaluada al terminar de cargar el archivo de audio.

path

Hay tres formas de indicar la dirección de un archivo de audio:

1- Si el archivo de audio se encuentra en el directorio sounds que se halla dentro del directorio SuperCollider:

"sounds/nombre_del_archivo"

2- Si el archivo se encuentra en algún subdirectorio de tu directorio de usuario:

"~/Documents/Audios/nombre_del_archivo".standardizePath

3- Si prefieres escribir la dirección completa:

"/Users/s/Documents/Audios/nombre_del_archivo"

```
(
b =                     // asignamos la variable b
                        // al objeto que será
                        // creado por la clase
                        // Buffer, para después poder
                        // mantener la comunicación
                        // con él

            Buffer.read(
                        s,      // el servidor
```

```
                    "sounds/a11wlk01-44_1.aiff" // la dirección del archivo de audio.
                                                // (este archivo de audio
                                                // viene con el programa)
                                                // generalmente vamos a
                                                // utilizar los valores
                                                // default de los demás
                                                // argumentos
                )
)
```

```
b = Buffer.read(s,"sounds/a11wlk01-44_1.aiff")
```

Si mandamos el mensaje .bufnum al objeto que acabamos de crear con Buffer, éste nos responderá el número de buffer que fue asignado a nuestro archivo de audio:

```
b.bufnum
```

Si no estamos seguros del número de canales de nuestro archivo de audio, podemos mandar el mensaje .numChannels a nuestro buffer.

```
b.numChannels
```

Para tocar nuestro buffer vamos a utilizar el UGen PlayBuf:

playbuf

Con PlayBuf leemos un sample que se encuentre cargado en la memoria.

```
PlayBuf.ar(numChannels, bufnum, rate, trigger, startPos, loop)
```

numChannels: número de canales del sample que va a ser leído. Hay que tener cuidado, ya

que si ponemos un número de canales erróneo, PlayBuf va a fallar silenciosamente, esto es, no va a marcar error, pero tampoco generará señal alguna.

bufnum: el número de buffer del sample.
rate: la velocidad a la que será reproducido el sample:

1.0: velocidad normal
0.5: mitad de velocidad, por lo que el sample sonará una octava abajo.
2.0: doble de velocidad (una octava arriba).
1.0: el sample será leído al revés, con la velocidad normal.

Si nuestro archivo de audio tiene una velocidad de *sampleo* distinta a 44100, para que la velocidad de reproducción sea la correcta tendremos que utilizar el UGen BufRateScale.

trigger: cuando recibe una señal que cambie de 0.0, o menos que 0.0, a mayor que 0.0, PlayBuf brinca a la startPos.

startPos: el número de cuadro en donde se iniciará la reproducción del *sample*. Cada segundo tienen 44100 cuadros (también conocidos como *samples*).

loop: si ponemos 1, cada vez que PlayBuf llegue al final del *sample* regresará al principio. Si ponemos 0, cuando llegue al final del *sample* se detendrá.

Primero cargamos un archivo de audio en la memoria:

```
b = Buffer.read(s, "sounds/a11wlk01-44_1.aiff")

(
{
var numeroDeCanales, numeroDeBuffer;
```

```
numeroDeCanales    = b.numChannels;    // al mandar el mensaje
                                       //.numChannels al objeto que
                                       // representa a nuestro buffer,
                                       // éste nos responderá
                                       // el número de canales del
                                       // archivo de audio que cargamos
                                       // en la memoria

numeroDeBuffer       = b.bufnum;       // con el mensaje .bufnum, pedimos
                                       // el número de buffer en donde se
                                       // encuentra el archivo de audio

PlayBuf.ar (numeroDeCanales, numeroDeBuffer)

}.scope
)
```

Evalúa las siguientes líneas:

```
b.bufnum

b.numChannels
```

Si quisiéramos ver toda la información del buffer, mandamos el mensaje .query:

```
b.query
```

Para que PlayBuf toque nuestro archivo de audio en *loop*, hay que asignar el valor 1 al argumento *loop*:

```
(
```

```
{
var numeroDeCanales, numeroDeBuffer;

numeroDeCanales = b.numChannels;
numeroDeBuffer = b.bufnum;

PlayBuf.ar(numeroDeCanales, numeroDeBuffer, loop: 1)

}.scope
)
```

Si queremos tocar nuestro archivo de audio a la mitad de velocidad, hay que poner 0.5 en el argumento rate:

```
(
{
var numeroDeCanales, numeroDeBuffer, velocidad;

numeroDeCanales = b.numChannels;
numeroDeBuffer = b.bufnum;

velocidad = 0.5;

PlayBuf.ar(numeroDeCanales, numeroDeBuffer, velocidad, loop: 1)

}.scope
)
```

Si queremos reproducir el sample al revés, a velocidad normal, ponemos -1.0 en el argumento rate:

```
(
{
var numeroDeCanales, numeroDeBuffer, velocidad;
numeroDeCanales = b.numChannels;
numeroDeBuffer = b.bufnum;

velocidad = -1.0;

PlayBuf.ar(numeroDeCanales, numeroDeBuffer, velocidad, loop: 1)

}.scope
)
```

Ahora vamos a controlar la velocidad con el mouse:

```
(
{
var numeroDeCanales, numeroDeBuffer, velocidad;

numeroDeCanales = b.numChannels;
numeroDeBuffer = b.bufnum;

velocidad = MouseX.kr(0.125, 2.0);

PlayBuf.ar(numeroDeCanales, numeroDeBuffer, velocidad, loop: 1)

}.scope
)
```

Vamos a utilizar un Impulse.kr como trigger para el argumento que lleva el mismo nombre. Cada vez que PlayBuf reciba un trigger, va a regresar al número de cuadro especificado en

startPos, cuyo valor default es 0 (el primer cuadro o sample del buffer).

```
(
{
var numeroDeCanales, numeroDeBuffer, velocidad, trigger;

numeroDeCanales = b.numChannels;
numeroDeBuffer = b.bufnum;

velocidad = 1.0;

trigger = Impulse.kr(MouseX.kr(0.7,4));      //vamos a mandar un trigger
                                             //cada segundo.

PlayBuf.ar(numeroDeCanales, numeroDeBuffer, velocidad, trigger)

// ya no va a ser necesario poner 1 en el argumento loop

}.scope
)
```

Con el mensaje .query podemos saber el número de cuadros o *samples* de nuestro archivo de audio:

```
b.query
```

Entre los datos que fueron desplegados en la Post Window vemos:

numFrames: 107520

Este es el número de cuadros de nuestro archivo.

Ahora vamos a hacer que cada vez que PlayBuf reciba un trigger, éste brinque a la mitad del buffer y no al principio de éste:

```
(
{
var numeroDeCanales, numeroDeBuffer, velocidad, trigger,
posicionInicial;

numeroDeCanales = b.numChannels;
numeroDeBuffer = b.bufnum;

velocidad = 1.0;

trigger = Impulse.kr(1);

posicionInicial = 107520/2;      // vamos a dividir el
                                 // el número total de
                                 // cuadros entre dos,
                                 // para obtener el
                                 // número de cuadro que
                                 // se encuentra a la
                                 // mitad del buffer

PlayBuf.ar(numeroDeCanales, numeroDeBuffer, velocidad, trigger, posicionInicial)

}.scope
)
```

En el ejemplo anterior escuchamos un *click* cada vez que el audio regresa a la mitad del buffer. Esto es causado por la diferencia entre la amplitud del audio antes de recibir el trigger y la amplitud después del trigger, después de que PlayBuf brincó a la mitad del archivo.

Para quitar el *click* tenemos que poner un envolvente a la señal, que inicie con una amplitud de cero y que regrese a cero justo antes de que el trigger cambié la posición del buffer.

2. reciprocal

```
(
{
var numeroDeCanales, numeroDeBuffer, velocidad, trigger,
posicionInicial, sig, env, duracionEnSegundos,frecuenciadeltrigger;

numeroDeCanales = b.numChannels;
numeroDeBuffer = b.bufnum;

velocidad = 1.0;

frecuenciadeltrigger = 10;

trigger = Impulse.kr(frecuenciadeltrigger);
posicionInicial = 107520/2;

duracionEnSegundos = 1/frecuenciadeltrigger;

/*como regresamos a la startPos antes de que PlayBuf haya  llegado al final del archivo
nuestro envolvente deberá durar el tiempo en segundos que haya entre dos triggers */

sig = PlayBuf.ar(numeroDeCanales, numeroDeBuffer, velocidad, trigger, posicionInicial);

env = EnvGen.kr(Env([ 0.0, 1.0, 1.0, 0.0 ], duracionEnSegundos * [ 0.025, 0.95,0.025 ]),
trigger);

/* también vamos a utilizar al trigger del PlayBuf, como trigger para el argumento gate del
```

```
EnvGen. EnvGen generará un envolvente cada vez que reciba un trigger */

sig * env
}.scope
)
```

BufFrames.kr

Este UGen calcula el número de cuadros de un buffer y con él es más fácil calcular la posición incial (startPos) de PlayBuf.

```
BufFrames.kr(bufnum)
```

bufnum: el número del buffer

Este UGen no nos va a dar directamente el número de cuadros de un buffer, sino que se lo comunicará a otros UGens.

Si quisieramos utilizarlo para que PlayBuf regrese al primer tercio del archivo de audio, tendremos que poner lo siguiente en el argumento startPos:

```
BufFrames.kr(bufnum) * (1/3)

(
{
var numeroDeCanales, numeroDeBuffer, velocidad, trigger, posicionInicial, sig, env,
duracionEnSegundos;

numeroDeCanales = b.numChannels;
numeroDeBuffer = b.bufnum;
```

```
velocidad = 1;

trigger = Impulse.kr(MouseY.kr(30,0.5));

posicionInicial = BufFrames.kr(numeroDeBuffer) * MouseX.kr(0,0.9999);

duracionEnSegundos = 1;

sig    = PlayBuf.ar(numeroDeCanales, numeroDeBuffer, velocidad, trigger, posicionInicial);

env          = EnvGen.kr(Env(     [ 0.0, 1.0, 1.0, 0.0 ], duracionEnSegundos * [ 0.05, 0.9,
0.05 ]), trigger);

sig * env

}.scope
)
```

Ahora vamos a utilizar el ratón para cambiar la posición inicial del archivo de audio y la velocidad de reproducción:

```
(
{
var trigger, sig, env, frecuencia = 6;

trigger = Impulse.kr(frecuencia);

sig = PlayBuf.ar(
                                b.numChannels,
                                b.bufnum,
                                MouseY.kr(-1.0, 1.0),
```

```
                        trigger,
                        MouseX.kr(0, BufFrames.kr(b.bufnum))
                    );

    env         = EnvGen.kr(Env(    [ 0.0, 1.0, 1.0, 0.0 ],
                                        recuencia.reciprocal * [ 0.05, 0.9, 0.05 ]),
                                    trigger);

sig * env

}.scope
)
```

Ahora con un SynthDef

```
a=Buffer.read(s,"sounds/a11wlk01-44_1.aiff")

a.numFrames

(
SynthDef(\sample,{|frecuencia=1,posc=0,trigger=1,rate=1|
    var sen,env;
    sen=PlayBuf.ar(1,a.bufnum,rate,Impulse.kr(frecuencia),posc);
    sen=sen*SinOsc.ar(MouseY.kr(20,1));
    sen=Pan2.ar(sen,LFNoise2.kr(10),0.9);
    env=EnvGen.kr(Env([ 0.0, 1.0, 1.0, 0.0 ],
                        frecuencia.reciprocal * [ 0.05, 0.9, 0.05 ]),
                        trigger,doneAction:2);
        Out.ar(0,sen*env)
        }).send(s)
        )
```

```
            Synth(\sample)
s.scope

(
Tdef(\hu,{var suma=0;
        inf.do{suma=suma+1;
        Synth(\sample,[\frecuencia,rrand(2,5),\posc,rrand(0,107520),
                        \rate,(suma%2+0.5)*1.2]);
            0.2.wait
            }}
            )
            )

            Tdef(\hu).quant_(0).play
            Tdef(\hu).stop



(
Tdef(\hu,{var suma=0;
        inf.do{suma=suma+1;
        Synth(\sample,[\frecuencia,0.1,\posc,suma%50*10,
                        \rate, (suma%10 -5)*0.5 ]);
        0.5.wait
        }}
        )
        )



(
Tdef(\hu,{var suma=0;
```

```
        inf.do{suma=suma+1;
        Synth(\sample,[\frecuencia,(suma%7+0.3)*1.3,\posc,suma%50*14,
                                \rate,1.3 ]);
        0.25.wait
        }}
        )
        )

        Tdef(\hu).stop

s.quit; //apagamos el servidor
```

Clase 9: Array

Por medio de los arrays podemos crear arreglos de objetos (recuerda que en Supercollider todo es un objeto) para luego poder utilizarlos según nuestras necesidades.

Definición:
Los array son colecciones de objetos indexados.

Ejemplo:
Escrbimos un array con los números del 1 al 5 y le adjudicamos la variable 'a'.

```
a = [1,2,3,4,5]
a
```

Ahora la variable 'a' será recargada con otro array.

```
a=[1,2,'tango',4,'ranchera']
a
```

Este manera de generar arrays nos exige detallar los objetos que seran creados. Otra manera de crear un array es por medio de los métodos que SuperCollider ofrece. Estos métodos son muy útiles y representan una manera más comoda de generar grandes colecciones de objetos; veamos algunos ejemplos:

Métodos

series (tamaño, comienzo, paso)

Crea una colección por medio de una serie aritmética.

```
Array.series(15,10,2);
Array.series(5,1,1);
[1,2,3,4,5]
```

.geom(tamaño, comienzo, paso)

Crea una colección por medio de una serie geométrica.

```
Array.geom(5, 1, 0.3);
```

.fib(tamaño,a, b)

Crea una colección por medio de la serie de Fibonacci.

```
Array.fib(36)
```

Array.fib(5,7,32)

.rand(tamaño, valor mínimo, valor máximo)

Crea una colección de valores random con un límite inferior y otro superior.

Array.rand(8, 1, 100);

.rand2(tamaño,límite)

Crea una colección de valores random con negativos y positivos.

Array.rand2(8, 100);

.linrand(tamaño, valor mínimo, valor máximo)

Crea una colección de valores random con límites inferior y superior en una distribución lineal.

Array.linrand(8, 1, 100);

.exprand(tamaño, valor minimo, valor maximo)

Crea una colección de valores random con límite inferior y superior en una distribución exponencial.

Array.exprand(8, 1, 100);

Instancias:

first: devuelve el primer valor de la colección.
Array.series(10,2,20).first

last: devuelve el ultimo elemento de la colección.
Array.series(10,2,20).last

choose: devuelve un elemento de la colección en random
Array.series(10,2,20).choose

wchoose: devuelve un elemento de la colección usando una lista de probablidades o pesos

Array.series(10,2,20).wchoose([0.3,0.02,0.08,0.1,0.1,0.05,0.1,0.1,0.05,0.05,0.05])
[0.3,0.02,0.08,0.1,0.1,0.05,0.1,0.1,0.05,0.05,0.05].sum

.sum: devuelve la suma total de valores dentro de un array.

```
Array.series(10,1,1).sum
```

.size: devuelve el número total de objetos comprendidos dentro del array.

```
Array.series(10,2,20).size
```

Matemáticas:

Los siguientes mensajes pueden ser enviados a un array.

```
[1, 2, 3, 4,-2].isNegative;
[1, 2, 3, 4].neg;
[1, 2, 3, 4].real;
[1, 2, 3, 4].reciprocal;
[1, 2, 3, 4].squared;
[1, 2, 3, 4].distort;
[1, 2, 3, 4].log10;
[1, 2, 3, 4].exp;
[1, 2, 3, 4].cubed;
[1, 2, 3, 4].midiratio;
[1, 2, 3, 4].midicps;
[1, 2, 3, 4].bilinrand;
[1, 2, 3, 4].tanh;
```

Observa la post

```
(
a=[2,3,4,24,10000,'si'];
Tdef(\no,{ var cont=0;
        inf.do{
        a [cont%a.size]).postln;
        cont=cont+1;
        0.3.wait
        }}
        )
)

Tdef(\no).play


Array.geom(15,0.1,1.16).pyramid
Array.geom(15,0.1,1.16).reverse
Array.geom(15,0.1,1.16).mirror
Array.geom(15,0.1,1.16).reverse.pyramid.mirror
```

Algunos ejemplos utilizando Arrays:

```
(
```

```
SynthDef(\sin,{|amp=0.8|
	var sen,env;
	sen=Pan2.ar(SinOsc.ar(1200,0,amp),0,0.9);
	env=EnvGen.kr(Env.perc(0,0.3),doneAction:2);
	Out.ar(0,sen*env)
	).send(s)
)

Synth(\sin)

// observa que valores devuelve el array .geom
// estos valores son aplicados al argumento amp
(
a=Array.geom(15,0.1,1.16).pyramid;
Tdef(\gi,{
	inf.do{
	Synth(\sin,[\amp,a.choose]);
	0.07.wait
	}}
	)
)
Tdef(\gi).play

// ahora utilizando una archivo de audio
// por medio de array modulamos la posc en el buffer y la amplitud
// observa el algoritmo n[n%n.size], donde n es el array

s.boot
a=Buffer.read(s,"sounds/a11wlk01-44_1.aiff")
a.numFrames
497114
(
SynthDef(\sample,{|posc,amp=0.1|
	var sen,env;
	sen=PlayBuf.ar(1,a.bufnum,1.6,Impulse.kr(1),posc)*amp;
	env=EnvGen.kr(Env([0,1,1,0],[1*0.025,1*0.95,1*0.025]),doneAction:2);
	Out.ar(0,sen*env)
	}).send(s)
)
Synth(\sample)

a=Array.geom(20,50,1.442)
a[4]
a=Array.series(10,50,12100).pyramid

a=Array.series(10,65050,4320)
```

```
(
a=Array.series(10,81050,3220).reverse;
b=Array.geom(15,0.1,1.16).reverse;
Tdef(\di,{var cont=0;
        inf.do{cont=cont+1;
        Synth(\sample,[\posc,a[cont%a.size],\amp,b[cont%b.size]]);

        0.12.wait
        }}
        )
)
Tdef(\di).play
```

Clase 10: Arrays

Un conjunto de elementos ordenados. Se crea escribiendo los elementos dentro de braquets y separandolos mediante comas.

```
[0,1,2,3]
```

Se pueden aplicar operaciones al array dando como resultado un nuevo array con la operación aplicada a cada elemento.

```
[0,1,2,3] + 1

[0,1,2,3] * 3
[0,1,2,3] / 2
```

Se pueden hacer operaciones entre arrays. Si los arrays tienen la misma cantidad de elementos, la operación se hace entre los elementos por pares uno a uno siguiendo el orden correspondiente. Es decir, el primer elemento de un array con el primer elemento del otro array, el segundo con el segundo, el tercero con el tercero, el cuarto con el cuarto, el quinto con el quinto, el sexto con el sexto, etc.

```
[0,1,2,3] + [3,2,1,0]
```

```
[0,1,2,3] * [3,2,1,0]

([0,1,2,3]*2) % ([0,1,2,3]+2)
```

Si los arrays son de diferente tamaño, el array mas pequeño vuelve a empezar desde su primer elemento y opera con el elemento que sigue del array más grande. El proceso termina con el último elemento del array de mayor tamaño.

```
[0,1,2,3] + [0,1]

[0,1,2,3] + [0,1] == [0+0, 1+1, 2+0, 3+1]
```

Los arrays responden a una gran variedad de métodos que nos sirven para manipular sus elementos. Aquí veremos sólo algunos que son de gran utilidad. Para conocer más sobre los métodos que se pueden aplicar a un array, revisar la ayuda de Array, ArrayedCollection y todas las Superclases subsecuentes. (Las Superclases son Clases que heredan todos sus métodos a otra clase mas específica).

```
[0,1,2,3].size // devuleve la cantidad de elementos del array
[0,1,2,3].scramble // desordena los elementos del array
[0,1,2,3].mirror // hace un espejo a partir del último elemento del array resultando un palindroma
[0,1,2,3].reverse // reordena los elementos escribiendolos en reversa
[0,1,2,3].choose // escoje un elemento del array
[0,1,2,3].wchoose([0.05,0.1,0.25,0.6]) // escoge un elemento del array según un peso probabilístico asignado a cada elemento. Estos pesos se escriben dentro de otro array que funciona como argumento del método wchoose. Los elementos de este array de pesos deben sumar 1
[0.1, 0.2, 0.3, 0.4].sum // suma entre si los elementos el array
[0,1,2,3].pyramid // hace una estructura piramidal con los elementos del array de la manera siguiente:
```

elemento 1,
elemento 1, elemento 2,
elemento 1, elemento 2, elemento 3,
elemento 1, elemento 2, elemento 3, elemento 4,
elemento 1, elemento 2, elemento 3, elemento 4, elemento 5
elemento 1, elemento 2, elemento 3, elemento 4, elemento 5, elemento6,...

[11,12,13,14].find([14]) // busca si algo es un elemento del array. Si lo es, devuelve el índice en donde se encuentra
[11,12,13,14].find([1]) // Si no se encuentra en el array entonces devuelve nil

g = [1,2,4,8] //Podemos asignar un array a una variable

g[0] // nos da el elemento del array g que se encuentra en el índice 0

g.add(3) // añade el objeto 3 como último elemento del array g. Notese que el array g sigue manteniendo la cantidad original de elementos mas el nuevo elemento. Si queremos añadir un elemento mas, usando por ejemplo g.add(10) no aumentará más el tamaño de g si no que sustituirá el elemento que habíamos añadido por el nuevo.

Observen:

g
g.add(100)
g.add(200)

Si queremos que se conserven los elementos añadidos y se sigan añadiendo mas, hay que hacerlo de esta forma:

g = g.add(100)
g = g.add(200)

Cualquier método que se pueda asignar a un número se puede asignar a un array de números.

```
[4.7, 5.6,6.5,7.4].asInteger
[0,2,4,5,7,9,11].midiratio
[60,62,64,65,67,69,71].midicps
[ 261.6255653006, 293.66476791741, 329.62755691287, 349.228231433,
391.99543598175, 440, 493.88330125612 ].cpsmidi

(0..10) // un atajo para escribir un array de números enteros consecutivos
```

También podemos usar la clase Array para generar ciertos tipos de arrays con estructucturas específicas.

```
Array.fill(4,"v") // hace un array con n cantidad de veces el elemento que queramos. (4 elementos "v")

Array.rand(4,2,5) // crea un array con n cantidad de elementos escogidos al azar de entre un rango determinado. (4 elementos escogidos al azar entre 2 y 5)
Array.series(10,0.1,0.1) // crea un array con n cantidad de elementos partiendo desde un número específico y haciendo una iteración que suma un incremento determinado. (10 elementos comenzando en 0.1 y añadiendo 0.1 en cada iteración)

Array.geom(10,0.1,0.1) // crea un array con n cantidad de elementos partiendo desde un número específico y hace una iteración que multiplica por un factor determinado. (10 elementos comenzando en 0.1 y multiplica por 0.1 en cada iteración)
```

Bueno, hay más métodos pero tenemos que detenernos para ver algunas aplicaciones.

Ejemplo 1

```
(
SynthDef(\asi, {|freq=1800, dur=0.01, amp=0.5|
                        var sig, env;
                        sig=SinOsc.ar(freq,pi/2,amp);
                        env=EnvGen.kr(Env.perc(0,dur),doneAction:2);
                        Out.ar(0,sig*env);
                        }).send(s);
)

Synth(\asi, [\amp, 1, \dur, 0.1])

~ritmo=[0,2,3,4,5,6,7];
~ritmo2=[0,2];
~ritmo3=[4,6]+24;
~ritmo4=[4,32+4,36+4];

(
Tdef(\mira_como_bailo, {var unidad=1.6;
                                inf.do{|i|
                                        if(~ritmo.find([i%8])!=nil, {Synth(\asi)});
                                        if(~ritmo2.find([i%8])!=nil, {Synth(\asi, [\freq, 200, \dur,
0.3, \amp, 0.25])});
                                        if(~ritmo3.find([i%32])!=nil, {Synth(\asi, [\freq, 250, \dur,
0.753, \amp, 0.25])});
                                        if(~ritmo4.find([i%64])!=nil, {Synth(\asi, [\freq, 60, \dur,
1.753, \amp, 0.25]);
                                                                Synth(\asi, [\freq, 130, \dur,
1.753, \amp, 0.25])});


                                        (unidad/16).wait;
```

```
                                        }
                                    }).quant_(0);
)

Tdef(\mira_como_bailo).play
Tdef(\mira_como_bailo).stop
```

Se pueden cambiar los valores de los arrays y se actualizan en tiempo real.

```
~ritmo=[0,3,6];
~ritmo2=Array.rand(4,0,7);
~ritmo3=Array.rand(8,0,32);
~ritmo4=Array.rand(6,0,32);
```

Ejemplo 2

```
~valores=[[1/4,0], [1/8,5], [1/8,6], [1/4,7], [1/4,8], [1,7],[1/4,12], [1/8,12], [1/8,8], [1/4,7], [1/4,8],
[1,7]]

(
Tdef(\mira_como_gozo, {var  tonalidad=440;
                                    inf.do{|i|

                                                Synth(\asi, [\freq, ~valores[i%~valores.size]
[1].midiratio*tonalidad, \dur, ~valores[i%~valores.size][0]]);
                                                (~valores[i%~valores.size][0]*1.6).wait;
                                                }
                                    }).quant_(0);
)

Tdef(\mira_como_gozo).play
```

```
Tdef(\mira_como_gozo).stop

~valores2=[[1/4,0], [1/4,-1], [1/4,0], [1/4,2], [3/4,3], [1/4,5], [3/4,3], [1/4,5], [3/4,3], [1/4, 2]]

(
Tdef(\mira_como_gozo_yo_tambien, {var  tonalidad=220;
                                  inf.do{|i|

                                          Synth(\asi, [\freq, ~valores2[i%~valores2.size]
[1].midiratio*tonalidad, \dur, ~valores2[i%~valores2.size][0], \amp, 0.75]);
                                          (~valores2[i%~valores2.size][0]*1.6).wait;
                                          }
                                  }).quant_(0);
)

Tdef(\mira_como_gozo_yo_tambien).play
Tdef(\mira_como_gozo_yo_tambien).stop
```

Junto los plays y los stops para poder dispararlos juntos

```
Tdef(\mira_como_gozo).play;
Tdef(\mira_como_gozo_yo_tambien).play;
Tdef(\mira_como_bailo).play;

Tdef(\mira_como_bailo).stop;
Tdef(\mira_como_gozo_yo_tambien).stop;
Tdef(\mira_como_gozo).stop;


(
```

```
SynthDef(\asado, {|freq=300, width=0.1, amp=0.5|
                                var sig, env;
                                sig=Pulse.ar(freq, width, amp);
                                env=EnvGen.kr(Env.sine(1,1),doneAction:2);
                                Out.ar(0,sig*env);
                                }).send(s);

)
Synth(\asado)
Synth(\asi)

~synths=[\asado, \asi]

Synth(~synths[2.rand])
```