4.3. Estructuras de programación orientada a objetos

La programación orientada a objetos (POO) es un modelo de programación, una forma particular de pensar y escribir el código, en la que se utilizan los elementos básicos vistos anteriormente: variables, expresiones y funciones..., pero organizados de modo más próximo al lenguaje verbal. El hecho de ser un modelo implica que tiene su propia lógica, y para usarlo no sólo debe comprenderse sino que hay que pensar el desarrollo del código según ese modelo, para poder aplicar después el lenguaje.

La POO nos hace pensar las cosas de una manera más abstracta y con una estructura más organizada, que sigue un proceso de descomposición del problema en partes bastante particular. Los métodos tradicionales de programación estructurada descomponen la complejidad en partes más simples y fáciles de codificar, de manera que cada sub-programa realiza una acción. En la POO no se descompone el problema en las acciones que tiene que hacer el programa, sino en objetos, por lo que debemos pensar en el posible escenario de esos objetos, cómo definir sus características y comportamiento. De ahí que el elemento básico de esta estructura de programación no son las funciones sino los **objetos**, entendidos como la representación de un concepto que contiene información necesaria (datos) para describir sus **propiedades** o atributos y las operaciones (métodos) que describen su **comportamiento** en ese escenario. Por eso implica una manera distinta de enfocar los problemas, que empieza por la **abstracción**, entendida en este caso como la capacidad mental para representar una realidad compleja en los elementos separados simplificados (objetos), ver aisladamente sus necesidades de definición y comportamiento, y también sus relaciones (**mensajes**) y comportamiento conjunto.

> Si nos detenemos a pensar sobre cómo se nos plantea un problema cualquiera en la realidad podremos ver que lo que hay son entidades (otros nombres que podríamos usar para describir lo que aquí llamo entidades son "agentes" u "objetos").
> Estas entidades poseen un conjunto de propiedades o atributos, y un conjunto de métodos mediante los cuales muestran su comportamiento. Y no sólo eso, también podremos descubrir, a poco que nos fijemos, todo un conjunto de interrelaciones entre las entidades, guiadas por el intercambio de mensajes; las entidades del problema responden a estos mensajes mediante la ejecución de ciertas acciones.[36]

Esta forma particular de estructurar el código favorece su reutilización al compartimentarse en módulos independientes. Recordad la cita de Katherine B. McKeithen[37] respecto a la importancia de organizar bien la información para procesarla como grupos significativos, algo así sucede en la POO.

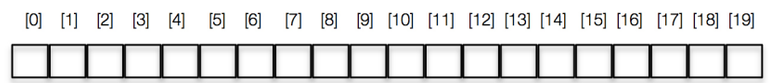
36 Luis R. Izquierdo (2007) "Introducción a la programación orientada a objetos", pág. 2. [texto on-line] http://luis.izqui.org/resources/ProgOrientadaObjetos.pdf [25.07.2012]..

37 Katherine B. McKeithen et al., "Knowledge Organization and Skill Differences in Computer Programmers" Cognitive Psychology 13(3), pág. 307. [Texto on-line] [Consultado: 12/05/2012] http://hdl.handle.net/2027.42/24336

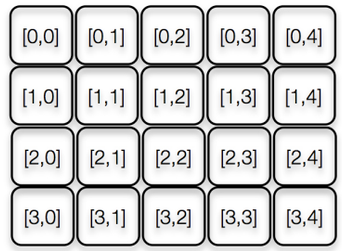
Nos centraremos en dos elementos que tienen un papel importante en este juego del pensar diferente, Tablas y Clases; señalando la particularidad de esos elementos en GAmuza, pues la hibridación de plataformas entre openFrameworks y Lua, plantean otra forma de aproximarse a la programación orientada a objetos.

4.3.1. Tablas

Las tablas mantienen cierta relación con la noción de array que en programación y matemáticas está vinculado a matriz, y significa un conjunto ordenado consecutivamente de elementos, del mismo tipo de datos, almacenados bajo el mismo nombre y para los que el sistema reserva memoria (igual que para las variables, pero aquí como una serie de variables relacionadas). El orden consecutivo asigna a cada elemento un ID, este ID es una forma numérica consecutiva que empieza por 0 y lo identifica de manera que se puede acceder a ese elemento en la programación individualmente. Los datos pueden ser de cualquier tipo pero no se pueden mezclar distintos tipos en un mismo array. Hay arrays de una dimensión (lineales) o de dos dimensiones (matrices) que podemos visualizarlo como una fila de contenedores que acogen cada uno de ellos un dato. En el siguiente ejemplo vemos cómo un array de 20 elementos se ordena hasta la posición 19, porque empieza a contar desde 0:



Los arrays de dos dimensiones constituyen una matriz de filas y columnas. Por ejemplo, la ventana de salida podemos verla como un array cuyo número de elementos es igual a los píxeles de la imagen y su dimensión es el resultado de multiplicar el ancho por el alto. Al igual que los array lineales, empieza a contar por cero y el primer elemento tiene el ID [0,0]

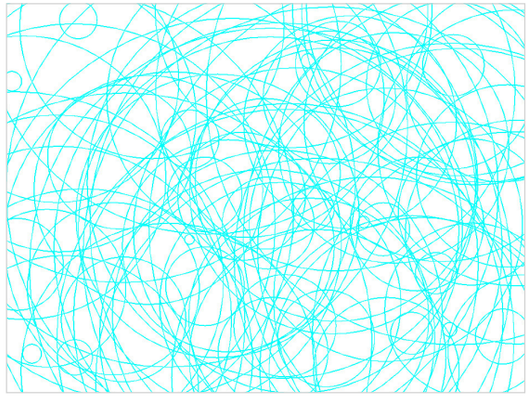


Teniendo en mente este concepto, vamos a ver cómo GAmuza no usa exactamente arrays, sino lo que en el lenguaje Lua se denominan **tablas** (librería table de Lua), pero puede hacer las mismas operaciones con mayor flexibilidad porque las tablas permiten utilizar datos de diferente tipo y simplifican mucho el proceso de programación. Además, las tablas en Lua no se entienden como lineales o de dos dimensiones, sino que una tabla lineal puede albergar múltiples dimensiones de datos.

La sintaxis para declarar una tabla es:

```
nombreTabla {}
```

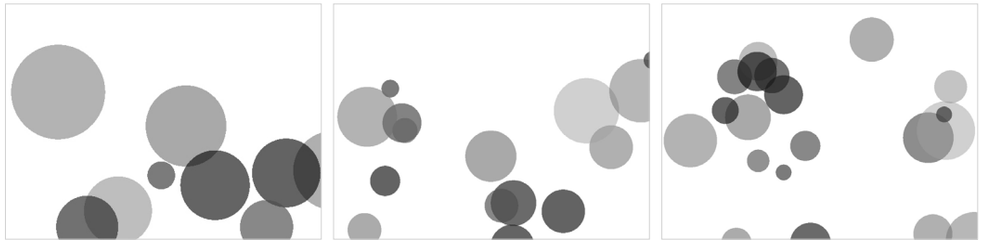
Para crearla pueden enumerarse los elementos que la constituyen entre {}, definir cual es el número de elementos por medio de una variable, o simplemente crear la tabla y todos los objetos utilizando una expresión `for` que recorrerá la tabla y asignará valores a todos los datos. Pasamos a describirlo comentado el código de algunos ejemplos.



```
/*
 GAmuza 043
 ---------------------
 Basics/tabla_circulosFijos.ga
 creado por n3m3da | www.d3cod3.org
*/


myTable = {}


function setup()
   for i = 0, 100 do                          // establece nº objetos del array
      myTable[i] = {}                         // crea otra tabla en la tabla
      myTable[i].x = ofRandomuf()*OUTPUT_W    // asigna valores cada coordenada x
      myTable[i].y = ofRandomuf()*OUTPUT_H    // asigna valores cada coordenada y
      myTable[i].diam = ofRandom(0,OUTPUT_W/2)  // y valores a los diámetros
   end
end

function draw()
   gaBackground(0.0,1.0)
   ofSetCircleResolution(60)
   ofSetColor(0,255,255)       // color cyan
   ofNoFill()                  // no rellenar
   for i = 0, 100 do           // dibuja los 100 círculos
      ofCircle(myTable[i].x, myTable[i].y, myTable[i].diam)
   end
end
```

La variable `i` de la estructura `for` va adoptando consecutivamente el ID de cada círculo de la tabla. Con el código `myTable[i] = {}`, crea una nueva tabla para cada uno de los objetos círculo, lo que permite definir sus propiedades sin hacer nuevos arrays, simplemente en el mismo `for` va asignando valores a cada una de esas propiedades utilizando el operador "`.`" junto al código que representa cada objeto en la tabla inicial: `myTable[i].`

En la función `draw()`, además de las funciones internas para el color y no relleno, otra estructura `for` dibuja todos los círculos recogiendo los datos de las propiedades del `setup()`. No hay función `update()` porque en el ejemplo las formas están fijas, no hay comportamiento. En el siguiente ejemplo sí se incluye este bloque con comportamientos, en el que también se incrementa el número de propiedades de cada uno de los objetos de la tabla.



```
/*
 GAmuza 043
 ---------------------
 Tabla círculos rebotan
 creado por mj
*/

circulos ={}                          // se declara la tabla
maxCirculos = 20                      // número máximo de objetos

function setup()
    ofEnableSmoothing()
    for i = 0, maxCirculos-1 do                       // recorre todos los objetos
     circulos[i] = {}                                 // crea otra tabla en la tabla
     circulos[i].posX = ofRandom(30, OUTPUT_W)        // coordenadas X al azar
     circulos[i].posY = ofRandom(OUTPUT_H/2)          // coordenadas Y al azar
     circulos[i].diam = ofRandom(15, 100)             // diámetro círculos
     circulos[i].velX = 0.000000                      // velocidad X
     circulos[i].velY = 0.000000                      // velocidad Y
     circulos[i].gY = 0.00184562                      // gravedad Y
     circulos[i].gX = ofRandom(-0.018326, 0.044326)   // gravedad X
     circulos[i].color= ofRandom(0, 100)              // gris azar para cada objeto
     circulos[i].transp = 220                         // transparencia
    end
end

function update()
   for i=0, maxCirculos-1 do
      circulos[i].posX = circulos[i].posX + circulos[i].velX  // posición + velocidad
      circulos[i].posY = circulos[i].posY + circulos[i].velY
      circulos[i].velX = circulos[i].velX + circulos[i].gX   // velocidad + gravedad
      circulos[i].velY = circulos[i].velY + (circulos[i].gY*circulos[i].diam)
```

```
      if circulos[i].posX <= 0 or circulos[i].posX >= OUTPUT_W-10 then
           circulos[i].velX = circulos[i].velX * -0.972773
      end                                  // invierte dirección al tocar los bordes
      if circulos[i].posY >= OUTPUT_H or circulos[i].posY <= 0 then
           circulos[i].velY = circulos[i].velY * -0.962722
      end
   end
end

function draw()
   gaBackground(1.0, 1.0)
   ofSetCircleResolution(60)
   for i= 0, maxCirculos-1 do          // dibuja todos los círculos
        ofSetColor(circulos[i].color, circulos[i].transp - circulos[i].diam)
         ofCircle(circulos[i].posX, circulos[i].posY, circulos[i].diam)
   end                           o
end
```

En el bloque `setup()` se asignan las **propiedades** de los objetos utilizando el operador "." junto al nombre de la tabla y el identificador del array de objetos que contiene `[i]`

```
nombreTabla[i].Propiedad
```

El `for` recorre el array, reserva la memoria que el sistema necesita para estos objetos y les asigna las propiedades definidas, que en este caso son: posición X y posición Y (para localizar el centro de cada circulo), radio (tamaño), velocidad X y velocidad Y (para el movimiento de los círculos) gravedad X y gravedad Y (para representar los rebotes y un movimiento menos lineal), color y transparencia.

Los valores de `i` van desde la posición `0` a la que define la variable `maxCirculos-1`, porque empieza a contar desde `0`. Los valores de las propiedades que se asignan inicialmente pueden después cambiar, pues un objeto puede tener distintos valores en momentos diferentes, pero siempre es el mismo objeto.

Los comportamientos se establecen en la función `update()`, donde se actualizan los datos de las variables que definen las propiedades. Con un nuevo `for` , a las coordenadas de posición de cada objeto se le suman las de velocidad, y a las de velocidad, las de gravedad en la coordenada de las X y las de gravedad por el diámetro del círculo en la coordenada Y, este incremento produce un movimiento más físico de peso del círculo. Después se ajustan los rebotes a los límites de la ventana mediante bloques condicionales `if`, invirtiendo la dirección del movimiento al multiplicar el valor de su velocidad por un número negativo con muchos decimales que no llega a 1. Finalmente, en el bloque `draw()` se dibujan todos los círculos mediante un nuevo `for`.

El siguiente ejemplo, más complejo, está basado en la noción de Random Walker, utiliza varios arrays y la función `ofNoise()` que se verá en la página 144.

```
/*
 GAmuza 043
 --------------------
 Shiffman_NatureOfCode/CH0_noiseWalk.ga
 creado por n3m3da | www.d3cod3.org
*/

walkers = {}
noiseTimeX = {}
noiseTimeY = {}
stepX = {}
stepY = {}
numW = 300
function setup()
   for i = 0,numW-1 do
        walkers[i] = {}
        walkers[i].x = OUTPUT_W/2
        walkers[i].y = OUTPUT_H/2
        noiseTimeX[i] = ofRandom(0,10000)
        noiseTimeY[i] = ofRandom(7000,14000)
        stepX[i] = ofRandom(0.001,0.003)
        stepY[i] = ofRandom(0.001,0.003)
   end
end

function update()
   for i=0,numW-1 do
        mapX = ofMap(ofNoise(noiseTimeX[i]),0.15,0.85,0,OUTPUT_W,true)
        mapY = ofMap(ofNoise(noiseTimeY[i]),0.15,0.85,0,OUTPUT_H,true)
        walkers[i].x = mapX
        walkers[i].y = mapY

        noiseTimeX[i] = noiseTimeX[i] + stepX[i]
        noiseTimeY[i] = noiseTimeY[i] + stepY[i]
   end
end
```

```
function draw()
   gaBackground(0.0,0.01)
   ofNoFill()
   for i=0,numW-1 do
      if math.fmod(i,7) == 0 then      // Devuelve el resto de la división de i por 7
                                       // y redondea el cociente a cero.
         ofSetColor(255)
      else
         ofSetColor(0)
      end
      ofRect(walkers[i].x,walkers[i].y,1,1)
   end
end
```

Algunas funciones internas vinculadas a las tablas[38]

`table.getn(myTable)` Para saber el número de elementos de una tabla

`table.foreach(myTable, myFunction)` Recorre cada índice de la tabla y le asigna la función (el segundo parámetro dentro del corchete).

`table.insert(myTable, [position], value) // ejemplo (myTable, 5, true)`

Esta función añade un valor al final de la tabla. Si se señala una posición exacta, para colocar el valor los índices siguientes se desplazan en consecuencia.

`table.remove(myTable, [pos])  // ejemplo (myTable, 3)`

Elimina un elemento de una tabla y mueve hacia arriba los índices restantes si es necesario. Si no se asigna posición borra el último elemento.

38 Más información en "Lua for Beginers" [texto on-line] http://lua.gts-stolberg.de/en/table.php [29.07.2012]

4.3.2. Clases

En una clase, el concepto más destacado es el de **objeto**, que representa cualquier cosa, real o abstracta, (componente conceptual del problema planteado). En la programación, un objeto se define en dos partes: 1) **Constructor**: una serie de datos que constituyen el estado particular (atributos o propiedades) del objeto y que se formalizan mediante un conjunto de variables. 2) **Update**: comportamiento propio de los objetos que se formaliza mediante **métodos** (funciones de objetos) relacionados con las variables.

Como se ha visto en los ejemplos anteriores, el objeto círculo tiene una serie **propiedades** declaradas con variables según lo que requiera la situación planteada: coordenadas del centro, radio o diámetro, color, transparencia, grosor del trazo..., de ellas, las que sean necesarias para definir el objeto serán los parámetros del constructor.

El **comportamiento** quedará regulado por los **métodos** que programemos para que esos objetos círculo sitúen sus coordenadas fijas o en movimiento, sigan una dirección de movimiento u otra, reboten con los límites de la ventana, etc.,

Definición de Clase

Estableciendo una relación algo simplista podemos entender las clases aludiendo a la Teoría de las ideas (o Teoría de las formas) de Platón. Las clases se corresponderían al mundo inteligible donde están las estructuras o modelos (las ideas) en las cuales se basan las cosas físicas concretas (los objetos) que podemos percibir en el mundo sensible.

Por ejemplo, la clase círculos puede contener todos los objetos circulares, con sus distintos tamaños, colores, etc., y sus distintos comportamientos. El objeto círculo es una **instancia** de la clase círculos. Cada objeto círculo tiene un estado particular independiente, pero todos tienen en común el hecho de tener una variable color, radio, etc. E igualmente deben tener en común la posibilidad de adoptar determinados comportamientos, por lo que la clase debe declarar o implementar los métodos que regulan el comportamiento de los objetos que contiene.

En resumen, una clase es como una plantilla que define las variables y los métodos que son comunes para todos los objetos que agrupa, en esta plantilla debe haber siempre un bloque dedicado al **constructor** que lleva el mismo nombre que la clase y define los parámetros necesarios para crear o instanciar un objeto de la clase.

Herencia

Es fácil comprender que la clase círculos pertenece a una clase mayor que alberga las figuras geométricas planas. Si en nuestro trabajo planteamos ambos niveles, un círculo concreto debe responder tanto a las propiedades de la clase círculos como a la de la clase figuras_planas, por lo

que compartirá esos atributos y comportamientos con objetos de otras posibles clases, como la clase triángulos o la clase cuadrados.

Esta herencia no limita las posibilidades de particularización de la clase círculos, más bien el orden de pensamiento es inverso al establecido aquí. La herencia permite definir nuevas clases partiendo de las existentes, de modo que heredan todos los comportamientos programados en la primera y pueden añadir variables y comportamientos propios que la diferencian como clase derivada o subclase. Y así se puede establecer un árbol de herencias tan ramificado como se necesite.

Mensajes

En la imagen anterior, además de ver un diagrama de herencia también muestra en una de las flechas que va de la clase triángulos a la clase rectángulos como se establece un mensaje. Es a través de los mensajes como se produce la interacción entre los objetos, permitiendo así programar comportamientos más complejos.

Por ejemplo, cuando un objeto triángulo cumple una determinada condición, puede enviar un mensaje a un objeto rectángulo para que realice alguno de los comportamientos que su clase (la del rectángulo) tiene definidos, en el mensaje se pueden incorporar determinados parámetros para ajustar el comportamiento.

Un mismo mensaje puede generar comportamientos diferentes al ser recibido por objetos diferentes. A esta característica se le denomina **polimorfismo**.

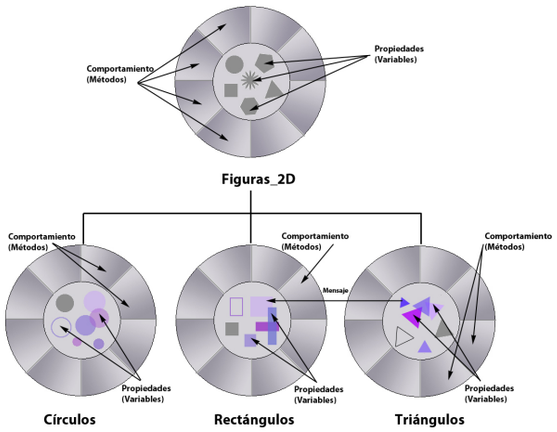
Es conveniente hacer diagramas de las clases con los atributos y comportamientos de los objetos antes de empezar a escribir el código, para respondernos a las preguntas:

¿Cuántas clases necesitamos definir y que herencia se establece entre ellas?

¿Dónde definimos cada propiedad?

¿Dónde definimos cada método?

¿Dónde implementamos cada método?



> Lua does not have a class system, but its powerful metaprogramming facilities makes defining classic objects straightforward.[39]

En el lenguaje Lua, una tabla también funciona como un objeto, porque las tablas tienen un estado y una identidad (`self`) que es independiente de sus valores, como hemos mencionado antes, dos objetos con el mismo valor son objetos diferentes, un objeto puede tener distintos valores en momentos diferentes, pero siempre es el mismo objeto. Al igual que los objetos, las tablas tienen un ciclo vital que es independiente de aquello que lo creó, o donde se crearon. Hemos visto que las tablas asignan propiedades a cada elemento mediante el operador `"."`, ahora veremos cómo se asocian métodos (funciones de objetos) a los objetos de una clase mediante el operador `":"`.

La sintaxis para declarar las clases es:

```
class ‘nombreClase’
```

Se hace el **constructor** con una función que lleva el nombre de la clase y se inicializa con

```
function nombreClase:__init (parámetro1, parámetro2,...)
```

Nombre de la clase, dos puntos `":"` y dos guiones bajos `"__"` seguidos de `init` con los parámetros que definen la construcción del objeto, entre paréntesis y separados por `","`.

La clase tiene su propio método `update()` que se escribe:

```
function nombreClase:update()
```

donde se pone el código para actualizar los valores de las variables y programar el comportamiento de los objetos.

Y su método `draw()` que contiene el código de las formas que va a dibujar.

```
function nombreClase:draw()
```

Después, se crean los objetos de la clase dándole los valores concretos a los parámetros definidos en el constructor.

```
ob1 = nombreClase(valorParámetro1, valorParámetro2,...)
```

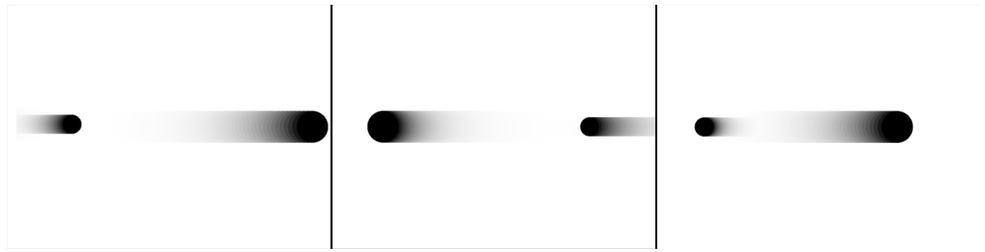
La herencia se implementada utilizando el concepto de metatablas, que actúan como superclases, y la función `setmetatable()`. El código `__index` almacena la metatabla de la tabla. En el siguiente código la tabla Rectangulo deriva de la tabla Formas[40]:

```
Rectangulo = {}
setmetatable(Rectangulo, {__index = Formas})
```

39 Lua-users wiki: Simple Lua Classes [texto on-line] < http://lua-users.org/wiki/SimpleLuaClasses > [25.07.2012]

40 Marcelo da Silva Gomes, "The Shape Example in Lua" [texto on-line] <http://onestepback.org/articles/poly/gp-lua.html> [25.07.2012]

Algunos ejemplos comentados de clases simples.



```
/*
 GAmuza 043
 ---------------------
 código básico para construir una clase.
 creado por mj
*/

// ------------------------------------------------
// ------- class foco
class 'foco'

//constructor
function foco:__init(x, d, s, g)
            //define parámetros y propiedades del objeto
   self.posX = x       // posición Y
   self.diam = d       // diámetro
   self.speed= s       // velocidad
   self.grav = g       // gravedad
end

function foco:update()        // actualiza parámetros para el comportamiento
   self.posX = self.posX + self.speed
   self.speed = self.speed + (self.grav*self.diam)

                        // condicionales: cuando llegue a los topes cambia de dirección
   if self.posX > OUTPUT_W - self.diam/2  then
      self.speed = self.speed * -0.999722
   end

   if self.posY < 0 + self.diam/2 then
      self.speed = self.speed * -0.999722
   end
end

function foco:draw()             // función para dibujar la forma del objeto
   ofCircle(self.posX, OUTPUT_H/2, self.diam)
end
```



```
//------ crear 2 objetos de la clase foco
foco1 = foco(25, 50, 3.5, 0.00184562)
foco2 = foco(125, 30, 1.5, 0.00184562)
//----------------------------------------

function setup()
    ofEnableSmoothing()
end

function update()
   foco1:update()
   foco2:update()
end

function draw()
   gaBackground(1.0, 0.1)
   ofSetCircleResolution(50)
   ofSetColor(0)
   foco1:draw()
   foco2:draw()
end
```

Después de **declarar** la clase: `class 'foco'` se define el **constructor**, para ello deben conocerse los parámetros necesarios para definir las **propiedades** concretas de los objetos que queremos crear. En el ejemplo, el código del **constructor** `function nombreClase:__init (parametro1, parámetro2,...)` corresponde a: `function foco:__init(x, d, s, g),` y dentro de él, utilizando el operador `"."`, se define cada propiedad de los objetos con la identidad (`self`).

Las clases tienen sus propios métodos `update()` y `draw()` para determinar los **comportamientos** y dibujar los objetos que se asocian a la clase mediante el operador `":"`

```
function foco:update()
function foco:draw()
```

Dentro del método `:update(),` para definir el comportamiento de movimiento, se suma el valor de la posición `X` a la velocidad, suma el valor de la velocidad al de la gravedad y se establecen las condiciones para cambiar el sentido de la velocidad cuando la coordenada `X` llega a los límites de la ventana. En el método `:draw()` se sitúan las funciones que dibujan los objetos.

Después de definir la clase se crean el objeto u objetos concretos, dando valores específicos a los parámetros determinados en el **constructor**. En este caso, los parámetros `(x, d, s, g)` para el objeto `foco1` son `(25, 50, 3.5, 0.00184562)`, es decir, el objeto aparece en la coordenada `x= 25`, tiene un diámetro de `50` píxeles, se mueve a una velocidad de `3.5` y con una gravedad de `0.00184562.`

En los bloques habituales de un programa en GAmuza, el código llama a los métodos de la clase.

En `function update()`

```
foco1:update()
foco2:update()
```

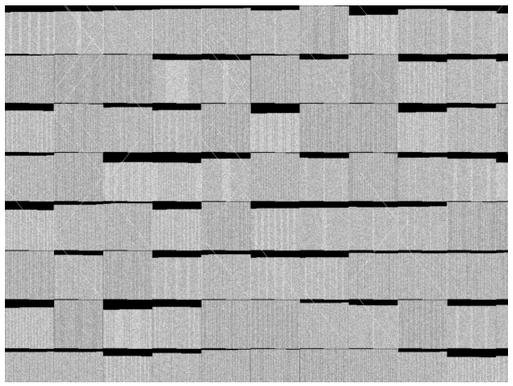
En `function draw()`

```
foco1:draw()
foco2:draw()
```

Es importante observar las diferencias entre clases y tablas Para una tabla las propiedades se definen en el `setup()` por el ID `[i]` utilizando un `for` para recorrer todos los objetos. Además la tabla no tiene métodos propios, por lo que actualiza su comportamiento en el bloque `function update()` y dibuja las formas en el bloque `function draw()` del programa. Cuando se quiere trabajar con muchos objetos (clase) controlando la posición de cada uno, se combina clase y tabla para asignar un id a cada objeto y poder programarle comportamientos específicos.

En el siguiente ejemplo se crea la clase, **'modulo'**. Todos los módulos se organizan siguiendo una estructura de tabla 2D, para configurar una retícula que ocupa toda la ventana de salida. Para determinar el número de objetos que contiene la tabla se debe asignar el id de cada módulo con un sólo número, y no 2, para ello se despliega un index continuo a partir del 2D. Esto permite recorrer todos los objetos con un doble for, manteniendo la condición 2D, pero trabajar con la tabla como si fuera de una dimensión.

```
/*
GAmuza 043
----------------------
clase y array 2D
creado por mj

*/

unit  = 100
ancho = math.ceil(OUTPUT_H/unit)
alto  = math.ceil(OUTPUT_W/unit)
contador = ancho * alto
mod ={}


// ----------------------------------
// ------- class modulo
class 'modulo'
      // declara la clase

function modulo:__init(mx, my, x, y, speed) // constructor
   self.mx = mx         //incremento posición x modulo
   self.my = my         //incremento posición y modulo
```




```
   self.x = x          //posicion inicial x
   self.y = y          //posicion inicial y
   self.speedx = speed                // velocidad x
   self.ydir= 1                       // direccion y
end

function modulo:update()
   self.x = self.x + self.speedx          // suma velocidad a la posicion

   if self.x >= unit or self.x <=0 then   // si x choca con los bordes del modulo
      self.speedx = self.speedx * -1      // invierte la velocidad
      self.x = self.x + (1 * self.speedx)
      self.y = self.y + self.ydir
   end

   if self.y >= unit or self.y <= 0 then      // si y choca con los bordes del modulo
         self.ydir = -1                       // hace lineas diagonales al final
         self.y = self.y + (1 * self.ydir)
   end
end

  function modulo:draw()
   ofSetColor(ofRandom(180,255), 90)     // color random casi blanco transparente
   ofCircle(self.mx + self.x, self.my + self.y, 1)  //dibuja puntos

end

////////////// FIN DE LA CLASE ///////////////

function setup()
   ofEnableSmoothing()

   index = 0
   for i = 0, alto-1 do  //recorre los objetos de la tabla 2D con un doble for
        for j = 0, ancho-1 do                          // asocia la tabla a la clase
           mod[index] = modulo(j*unit, i*unit, unit/2, unit, ofRandom (0.6, 0.8))
                                  //asigna valores a los parámetros del constructor
           index = index + 1     // designa el n° de objetos = ancho * alto (contador)
        end
   end
end

function update()
   for i = 0, contador-1 do
         mod[i]:update()       //aplica el update() de la clase a la tabla
   end
end

function draw()
   //gaBackground(0.0, 1.0)
   for i = 0, contador-1 do
        mod[i]:draw()          //aplica el draw() de la clase a la tabla
   end
end
```