Tutorial de introducción a la programación con Processing

Escuela de Verano “Laboratorio de lo intangible”

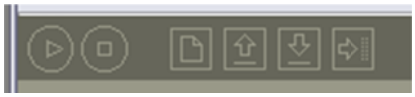
Zaragoza, 3-12 Julio 2017

Secciones:

Introducción

Interfaz de Processing:

La Interfaz de Processing es sencilla. Tiene 6 opciones: run, stop, new, open, save, export.



Cuando creamos un proyecto nuevo en Processing, creamos un **Sketch**, y el **Sketch folder** es la carpeta donde éste se guarda. En principio, el Sketch consta tan solo de la misma carpeta y de un archivo **.pde** que contiene el código. Luego se pueden añadir carpetas para archivos de imagen, video, etc.

RUN: Sirve para ejecutar el código. Con la magia de este simple botón nos ahorramos lo que en programación se conoce como compilación.

STOP: Para el programa

NEW: Crea un Sketch nuevo

OPEN: Abre un Sketch existente

SAVE: Guarda el Sketch con el que se está trabajando. Hay que utilizarlo mucho, por si acaso, pero aún más utilizar el "Save as" (ctrl+shift+s) para ir guardando versiones mientras trabajamos en nuestros proyectos.

EXPORT: Sirve para preparar el sketch para ser ejecutado a través de un navegador, ya sea en la web o simplemente fuera del entorno Processing.

FILE -> PREFERENCES: Cuatro opciones muy útiles: donde se guardan los sketch, tamaño de fuente en la interfície...

SKETCH -> PRESENT: Ejecutar el programa ocultando todo en la pantalla excepto la ventana de Processing. Esta es la respuesta a: "como se ejecuta un sketch de Processing a pantalla completa?".

TOOLS -> AUTO FORMAT: Da un formato al código que facilita seguir su lógica, en tanto que lo alinea verticalmente (nada como probarlo para entender de qué estoy hablando!). En versiones paleontológicas del programa esta opción se llamaba "Beautify"!

TOOLS -> CREATE FONT: Prepara un archivo fuente para ser utilizado, y nos ahorra tener que buscarlo por nuestra máquina o por la web.

TOOLS -> COLOR PICKER: Un selector de color que permite visualizar, en un color cualquiera, sus valores RGB o HSB para así utilizarlos en el código.

"Hello World!"

Processing, como cualquier entorno de programación, tiene una consola. La consola de Processing se encuentra debajo, en la interfaz de Processing, con fondo negro.

Aquí es donde veremos los resultados de nuestro programa, lo que en el argot de los programadores se conoce como el “Hello World!”.

Ejecutarlo, en Processing, es tan sencillo como escribir:

```
print("hello world!");
```

Habréis notado que cada vez que ejecutábamos el programa se abría una pequeña ventana en blanco (en gris, de hecho). Se trata de la ventana que Processing crea y es donde nos centraremos, sobretodo, a partir de ahora. Sin dejar nunca de tener un ojo a la consola, herramienta esencial para el programador, evidentemente lo que nos interesará es la ventana, ya que allí estará el resultado final de nuestra aplicación.

Funciones gráficas

Para empezar a ver de qué estamos hablando, ya que Processing es un entorno esencialmente gráfico, vamos a realizar otro tipo de "hello world!" haciendo un primer dibujo. Lo primero que vamos a hacer es dibujar una línea con:

```
line(20,20,80,80);
```

Ahora sí que donde hay que mirar es a la ventana gris. Si cambiáis éste código por el que sigue

```
line(40,15,90,95);
```

y comparáis los resultados podéis empezar a suponer cómo está respondiendo el programa a nuestras instrucciones en este caso.

COLOR

Cuando trabajamos con Processing, el ordenador entiende el color con cuatro valores (rojo, verde, azul y alfa) ocupa 8 bits (un byte) en memoria. Un byte tiene 256 valores posibles. Así pues, el color en Processing tendrá un rango de 0 a 255 por cada uno de estos cuatro valores.

El valor de un color en Processing se expresa así: **color(255,0,0)** pera un rojo puro, por ejemplo, o para un verde con alfa (transparencia) del 50%: **color(0,255,0,127)**. Si no se especifica el parámetro alfa, se entiende que su valor es máximo, 255, y que por lo tanto no hay transparencia (la opacidad es total).

Hay varias maneras de utilizar el color. De momento, veremos las funciones **background(), stroke()** y **fill()**. Dentro de sus paréntesis (es decir, como parámetros), pondremos el valor de color (en forma de uno, tres, o cuatro números de 0 a 255), para cambiar, respectivamente, el color de fondo de una ventana, el color del trazo en una forma geométrica, o su color de relleno.

Por ejemplo:

```
background(0); //para un fondo negro

stroke(255,0,0); //para un trazo rojo.

fill(255,255,0,32); //para un color de relleno amarillo con muy poco alfa.
```

Funciones de dibujo I

Volvemos ahora al “hello world” gráfico:

Escribid

```
line(20,20,80,80);
```

en la interfaz de Processing y clicad *Run.*

Veréis que una línea negra en la ventana gris que Processing ha abierto.
Una línea que va del punto 20,20, al 80,80 en una cuadrícula de 100×100 píxeles.

Si escribís

```
rect(20,20,60,60);
```

veréis, efectivamente, que se crea un rectángulo (cuadrado en este caso) a partir del punto 20,20 y de 60 píxeles de largo y ancho.

Y así sucesivamente con las siguientes formas geométricas:

line() p.e.: **line(30, 20, 85, 75);**

rect() p.e.: **rect(30, 20, 55, 55);**

ellipse() p.e.: **ellipse(56, 46, 55, 55);**

triangle() p.e.: **triangle(30, 75, 58, 20, 86, 75);**

Para combinar esto con las funciones de color, hay que llamar las funciones de color ANTES de dibujar, como si de coger un lápiz de tal o cual color se tratara. Así:

```
fill(0,255,0);
stroke(255,0,0);
rect(30, 20, 55, 55);
```

dibujará un rectángulo verde con el borde rojo. Estos colores se mantendrán en la paleta hasta que llamemos de nuevo las funciones fill y stroke. Background, por su parte, llenará toda la ventana del color especificado (sin admitir alfa), con lo que hay que escribir esta función antes de cualquier dibujo.

Funciones de dibujo II

size();

Cambia el tamaño de la ventana, que por defecto es de 100×100. size(x,y);
size() puede recibir dos o tres parámetros. Si recibe dos (o los dos primeros en caso de recibir tres), se refieren al tamaño de la ventana en píxeles. (El tercero, que por ahora podéis ignorar, se refiere al motor de “rendering”, de dibujo).

Por ejemplo:

```
size(400,300);

size(423,67);
```

background();

Otra instrucción útil para dibujar, y siempre, es background(), que le dice a Processing cual tiene que ser el color de fondo de una ventana.

```
background(127);

background(127,192,255);
```

Esta instrucción la debemos situar justo detrás de size() o, en todo caso, antes de hacer ningún dibujo, ya que de hecho tapa todo lo que hay en la ventana con el color especificado. Por defecto, su valor es el gris que hemos visto en los primeros ejemplos.

fill();

Como su nombre indica, esta instrucción especifica el color con el que va a rellenar la próxima forma geométrica que se dibuje. Los parámetros (valores entre paréntesis) especifican el color y pueden ser uno, dos, tres o cuatro.

```
fill(153);

fill(255,0,127);
```

Fill afectará todos los dibujos que se realicen DESPUÉS en el código. No sólo el siguiente, y no afecta para nada a los anteriores.

stroke();

Lo mismo que fill() pero afecta el contorno, el trazo de las formas y las líneas.

```
stroke(153);

stroke(255,0,127);
```

Stroke afectará todos los dibujos que se realicen DESPUÉS en el código. No sólo el siguiente, y no afecta para nada a los anteriores.

strokeWeight();

Recibe un parámetro. Especifica el grosor del trazo en píxeles. Por defecto, es uno.

```
strokeWeight(3);
```

noFill();

Especifica que a partir de este punto no se rellenen las formas.
No recibe parámetros. Es decir, no hay que poner nada entre los paréntesis. Ningún valor. Simplemente se invoca la función.

```
noFill();
```

noStroke();

Especifica que a partir de este punto no se dibujen los trazos.
No recibe parámetros. Es decir, no hay que poner nada entre los paréntesis. Ningún valor. Simplemente se invoca la función.

```
noStroke();
```

Ejemplos de código de dibujo

Visto esto, ya tenemos elementos para entender código un poco más complejo. Copiad los siguientes tres ejemplos a Processing, e intentad seguir la lógica del código y predecir qué va a pasar una vez cliquéis RUN.

Ejemplo 1:

```
size(300,300);
background(255);
fill(127,255,0);
stroke(255,0,0);
rect(50,50,200,200);
```

Ejemplo 2:

```
size(200,200);
background(0);
fill(0,0,255);
stroke(255,0,0);
ellipse(100,100,110,50);
fill(0,255,0);
rect(100,100,40,35);
```

Ejemplo 3:

```
size(600,400);
background(0);
fill(255,0,0);
strokeWeight(5);
stroke(255);
ellipse(250,150,200,180);
noStroke();
fill(0,255,0,127);
rect(250,150,200,120);
stroke(255,200);
line(100,100,350,210);
```

Declaración de variables y constantes

Veremos en un momento cuales son los tipos de variables accesibles, pero por ahora hay que tener claro que en Processing **hay que declarar las variables y las constantes**. Esto significa que antes de darle un nombre y un valor, a una variable hay que decirle de qué tipo de datos va a constar. Para una variable del tipo Entero:

```
int miDis = 445;
```

declara una variable con el nombre "miDis", el valor 445, y del tipo de dato entero. **Al igual que el nombre, el tipo de dato no se puede cambiar**.

Una vez declarada una variable, la podemos utilizar:

```
int miDis = 445;
print(miDis);
```

Lo que sí que se cambia, pues, es el valor.

```
int a = 78;
println(a);
a = 99;
println(a);
```

y se pueden hacer operaciones entre variables:

```
int a = 78;
float b = 13.46;
println(a*b);
```

Y esto tiene también su aplicación en programación visual. Así, si dibujo un cuadrado, puedo tener como variables su posición y medidas:

```
int x = 78;
int y = 121;
int tamano = 50;
rect(x,y,tamano,tamano);
```

Tipos de datos

El valor de variable debe corresponder a uno de los tipos de datos con que el programa puede trabajar. Más concretamente, debe corresponder con el de la variable en cuestión. Los más comunes en Processing son **int** (números enteros), **float** (número con decimales), **boolean** (verdadero o falso), **color** (que de hecho es una estrucutra compleja formada por quatro enteros) y **string** (cadena de carácteres).Obviamente están en la referencia: **int, float, boolean, color, String**

Estos y otros tipos de datos se pueden utilizar como se ve en el ejemplo siguiente:

```
boolean a; // true or false
byte b; // -128 to 127
char c; //carácter
color d;
float e; //número con decimales
int f; // entero
String g; //cadena de carácteres

a = false;
println(a);

b = -32;
println(b);

c = 'f';
println(c);

d = color(233,127,23);
println(d);

e = 983.243812;
println(e);

f = 78;
println(f);g = "ceci nest pas une pipe";
println(g);
```

Variables de sistema

Las variables de sistema no dejan de ser variables. Es decir, nos referimos a ellas por su nombre y contienen un valor. Su característica especial es que no las tenemos que declarar, ya que, al ser de sistema, son accesibles y calculadas automáticamente para nosotros. Por ahora, nos interesará ver estas variables de sistema: **width** : ancho de la ventana, **height** : alto de la ventana, **mouseX** : posición del ratón en el eje X, **mouseY** : posición del ratón en el eje Y.

Con **with** y **height**, nos ahorramos de ir constantemente a la función **size()** para recordar qué tamaño de ventana estamos utilizando. Y aún más importante, si utilizamos **with** y **height** podemos cambiar el tamaño de la ventana sin alterar la posición y tamaño relativos de los objetos:

```
size(300,300);

ellipse(width/2,height/2,width-20,height-20);
```

Si cambiamos la primera línea de código, no hace falta tocar la segunda:

```
size(120,120);

ellipse(width/2,height/2,width-20,height-20);
```

Setup y Draw: Processing en movimiento

Si hay algo muy específico de Processing (y que pone especialmente nerviosos a algunos programadores de verdad) son el void setup() y el void draw().

Lo importante es que **setup** y **draw** son las dos funciones principales de Processing. Las que se ejecutan siempre. Primero **setup**, una sola vez, y luego **draw** repetidamente. Esto es lo que rompe la linearidad estricta en la lectura y ejecución del código, y lo que nos permite hacer cosas mucho más interesantes que simples dibujos.

Hasta ahora las hemos ignorado porque el código que utilizábamos se ejecutaba una sola vez, de arriba a abajo. Ahora, todo lo que esté dentro de **draw** se va a ejecutar repetidamente, también de arriba a abajo, hasta que paremos el programa (también hay la opción de pararlo con código, pero por ahora lo ignoramos). Con los dos ejemplos que siguen debería verse claro.

```
int numero;

void setup(){
numero = 13;
}

void draw(){
numero = numero + 1;
println(numero);
}
```

y

```
float pos;

void setup(){
size(400,100);
pos = 13;
}

void draw(){
pos = pos + 0.1;
ellipse(pos,50,20,20);
}
```

y si retomamos el último ejemplo de las variables de sistema, con la elipse, podemos ver como combinando éstas con una variable convencional y con **setup()** y **draw()** podemos por fin tener una cierta (aunque básica) interactividad:

```
int sz = 30;
```

```
void setup(){
size(200,200);
}

void draw(){
ellipse(mouseX,mouseY,sz,sz);
}
```

Y finalmente, añadiendo una sola línea, borramos en cada *frame* el dibujo anterior para dar la sensación de movimiento:

```
int sz = 30;

void setup(){
size(200,200);
}

void draw(){
background(0);
ellipse(mouseX,mouseY,sz,sz);
}
```

Estructuras condicionales

Lo que hacen los condicionales es comprobar si tal o cual condición se cumple o no. Si se cumple el programa realizará una acción, y si no, realizará otra o nada. Es decir: si se cumple A, entonces haz B. O en inglés: **if a, then b**.

```
if(condicion){
hazTalCosa();
}

if(condicion){
hazTalCosa();
} else {
hazTalOtra();
}
```

Respecto a las condiciones, en general, éstas estarán compuestas por comparaciones entre valores. Si "a" es mayor a "b", haz X. Si "c" es menor o igual a "d", haz Y. Si "e" no es igual a "f", haz Z, etc.

```
if(a > b){
hazTalCosa();
}
```

Operadores

Los operadores que podemos utilizar son:

== (igualdad)

```
if(a == b){
hazTalCosa();
}
```

!= (no igualdad)

```
if(a != b){
hazTalCosa();
}
```

> (mayor a)

```
if(a > b){
hazTalCosa();
}
```

Operadores lógicos

Los operadores lógicos son:

&& ("y" lógico)

El "y" lógico compara dos valores booleanos y devuelve verdadero solamente si los dos valores comparados son verdaderos.

```
true && true = true
true && false = false
false && true = false
false && false = false
```

```
if((a < b)&&(c==d)){
hazTalCosa();
}
```

|| ("o" lógico)

El "o" lógico compara dos valores booleanos y devuelve verdadero si uno de los dos valores comparados son verdaderos.

```
true || true = true
true || false = true
false || true = true
false || false = false
```

```
if((a < b) || (c>d)){
hazTalCosa();
}
```

! ("no" lógico)

El no lógico no compara valores sinó que lo que hace es invertirlos.

```
!false = true
!true = false
```

```
if(!a){
hazTalCosa();
}
```

Ejemplos

Podéis cambiar los valores true/false de los booleanos para entender cómo funcionan el "y" y el "o" lógicos:

```
boolean condUno = true;
boolean condDos = false;

//comprobamos el "o" lógico
if(condUno || condDos){
print("se cumple el ||");
}
else{
print("no se cumple el ||");
}

//comprobamos el "i" lógico
if(condUno && condDos){
print("se cumple el &&");
}
else{
print("no se cumple el &&");
}
```

Con un elemento más gráfico, podemos utilizar un condicional para decidir cuándo se para una acción:

```
float pos;

void setup(){
size(300,100);
pos = 13;
}

void draw(){
if(pos <= 200){
pos = pos + 4;
}
ellipse(pos,50,20,20);
}
```

(ver código Fuente: **globus)**

```
float tamano = 1;

void setup(){
  //tamaño
  size(200,150);
}
void draw(){
  //el fondo, en cada frame
```

```
  background(0);
  //el dibujo
  ellipse(width/2,height/2,tamano,tamano);

  //CONDICIONAL: SOLO CRECE SI ES MENOR A 150
  if(tamano < 120){
    tamano = tamano + 0.5;
  }

  if(tamano >= 120) {
    tamano = 1;
  }
}
```

O lo podemos aplicar a la posición:

```
float pos;

void setup(){
size(300,100);
pos = 13;
}

void draw(){
if(pos <= 200){
pos = pos + 4;
}
ellipse(pos,50,20,20);
}
```

(ver código Fuente: **bolaStop)**

```
El condicional nos permite parar la pelota a partir
 de cierto punto
 */

float pos;

void setup(){
  size(300,100);
  pos = 13;
}

void draw(){
  if(pos <= 200){
    pos = pos + 4;
  }
  ellipse(pos,50,20,20);

  if(pos >= 200){
     pos = 13;
     background(127);
  }
}
```

O utilizarlos para hacer "rebotar" la pelota en los bordes de la ventana y simular así un espacio cerrado:

```
int pos, vel;
int sz = 20;

void setup(){
size(300,200);
pos = width/2;
vel = 5;
}

void draw(){

background(0);

//actualitzamos la posición
pos = pos+vel;

//dibujamos
ellipse(pos,height/2,sz,sz);

//comprobamos si está en los límites de la ventana
//si lo está, invertimos el signo de la velocidad:
if((pos<0)||(pos>width)){
vel = -vel;
}
}
```

(ver código fuente: **bouncingBallH)**

```
El condicional nos permite hacer rebotar la pelota
 */

 int pos, vel;
int sz = 20;

void setup(){
  size(200,150);
  pos = width/2;
  vel = 5;
}

void draw(){

background(0);

  //actualitzamos la posición
  pos = pos+vel;

  //dibujamos
  ellipse(pos,height/2,sz,sz);

  //comprobamos si está en los límites de la ventana
  //si lo está, invertimos el signo de la velocidad:
  if((pos<0)||(pos>width)){
```

```
    vel = -vel;
  }
}
```

Más ejemplos

Partiendo de lo visto hasta ahora, es fácil añadir el rebote para el eje Y y crear otro ejemplo típico de programación: *the bouncing ball*.

Ver código fuente: **bouncingBall**

```
int posX, posY, velX, velY;
int sz = 20;

void setup(){
  size(200,150);
  posX = width/2;
  posY = height/2;
  velX = 5;
  velY = 5;
}

void draw(){
  background(0);

  //actualitzem posicions
  posX = posX+velX;
  posY = posY+velY;

  //dibuixem
  ellipse(posX,posY,sz,sz);

  //comprovem la posició X
  if((posX<0)||(posX>width)){
    velX = -velX;
  }
    //comprovem la posició Y
  if((posY<0)||(posY>height)){
    velY = -velY;
  }
}
```

Y finalmente, también podemos utilizar los condicionales para hacer crecer y decrecer la pelota, ya sea para que parezca que respire, o para simular un eje Z.

(ver código fuente: **breathBall)**

```
Utilizamos aquí un codicional para hacer "respirar"
 nuestra amiga la pelotita
 */

int tamanoMinimo = 20;
int tamanoMaximo = 175;
int intervalo = 3;
boolean creciendo = true;
```

```
int tamanoPelota = 2;

void setup(){
  size(200,200);
  smooth();
  strokeWeight(3);
}

void draw(){
  background(255,255,127);
  //colores
  fill(255,127,255);
  stroke(127,255,255);

  //El tamaño de la pelota debe cambiar según estemos creciendo
  //o decreciendo:
  if(creciendo==true){
    tamanoPelota = tamanoPelota + intervalo;
  }
  else { //es decir, si no está creciendo
    tamanoPelota = tamanoPelota - intervalo;
  }

  ////////////////////////
  //Y aquí hay que comprobar si se llegó a un tamaño límite,
  //para cambiar el ciclo crecer decrece
  if(tamanoPelota > tamanoMaximo){
    creciendo = false;
  }
  if(tamanoPelota < tamanoMinimo){
    creciendo = true;
  }

  //Finalmente, dibujamos la pelotita
  ellipse(width/2,height/2,tamanoPelota,tamanoPelota);
}
```

Más ejemplos

Los condicionales evalúan una condición (lo que va entre paréntesis) y ejecutan el código que escribimos entre las claves según si ésta se cumple o no.En este otro ejemplo, el color de **stroke** y **fill** del cuadro cambia, vía condicionales, según la posición del mouse.

(ver código fuente: **quad1)**

```
void setup(){
  size(200,200);
  strokeWeight(20);
}

void draw(){

  //primer lliguem el FILL a l'eix horitzontal del mouse
  if(mouseX < width/2){ //si el mouse estÃ  al cantÃ³ esquerre...
    fill(255,0,0); //preparamos un fill rojo
  }
  else { //si estÃ  a la dreta
    fill(0,255,0); //preparamos un fill verde
```

```
  } //i segon, fem que l'STROKE depengui de l'eix vertical del mouse
  if(mouseY < height/2){ //si el mouse estÃ  a la part de dalt
    stroke(255,255,0); //preparem un stroke groc
  }
  else { //si estÃ  cap a baix
    stroke(0,0,255); //preparem un stroke blau
  }

  //i finalment dibuixem el quadre, amb els color que hem carregat a
fill i stroke
  //segons la posiciÃ³ del mouse
  rect(50,50,100,100);
}
```

```
 //primero ligamos el FILL al eje horizontal del mouse
           if(mouseX < width/2){ //si el mouse está hacia la izquierda
             fill(255,0,0); //preparamos un fill rojo
           }
           else { //si está hacia la derecha
             fill(0,255,0); //preparamos un fill verde
          }

 //y segundo, hacemos que el STROKE dependa de eje vertical del mouse

           if(mouseY < height/2){ //si el mouse está hacia la
izquierda

             stroke(255,255,0); //preparamos un stroke amarillo

           }

           else { //si está hacia la derecha

             stroke(0,0,255); //preparamos un stroke azul

           }
```

En el ejemplo que sigue, un **if, else** nos sirve para dibujar el globo, que estará o no creciendo, o bién un recuadro rojo si el globo ha explotado.

También en éste ejemplo vemos una nueva **variable de sistema: mousePressed**, que, como su nombre indica, nos dice si el mouse está siendo clicado o no. Pertenece al tipo de datos **booleano**.

(Ver código fuente: **pinxo)**

```
int sz = 20; //mida del globus: inicialment 20
int creixement = 1; //variable de creixement del globus
int midaPunxa = 35; //mida de la punxa
```

```
void setup(){
  size(200,200);
  fill(255,255,127);
  smooth();
  stroke(255,127,127);
}

void draw(){
  background(0);

  //dibuixem la punxa:
  stroke(255,127,127);
  line(0,height/2,midaPunxa,height/2);

  if(mousePressed){ //si estem clicant
    sz = sz + creixement; //size creixerÃ  segons la variable
"creixement"
  }

  /*abans de dibuixar el globes, cal comprobar si toca o no la punxa.
   Sabem que el globus toca la punxa si la meitat del seu diÃ metre
   Ã©s menor a la meitat de WIDTH menys la mida de la punxa */

  if(sz/2 > (width/2)-midaPunxa){
    //com que ha tocat la punxa, dibuixem el recuadre vermell:
    fill(255,0,0);
    rect(0,0,width,height);
  }
  else { //com que no el toca, dibuixem la bola
    //primer el globus
    //sense stroke
    noStroke();
    ellipse(width/2,height/2,sz,sz);
    //println(sz);
  }

}
if(sz/2 > (width/2)-tamanoPincho){
      //como ha tocado el pincho, dibujamos un recuadro rojo:
      fill(255,0,0);
      rect(0,0,width,height);
    }
    else { //como no lo toca, lo dibujamos
      //dibujamos el globo:
      //sin stroke:
      noStroke();
      ellipse(width/2,height/2,sz,sz);
      println(sz);
    }
```

Y uno más: Un simple **if** nos permite saber si la bola toca o no el obstáculo. Esto sí, con la inestimable ayuda de **abs()**, que convierte cualquier valor en su valor absoluto (es decir, *siempre positivo*, como decía Louis).

(Ver código fuente: **bballObs)**

```
float posX, posY, velX, velY;
int sz = 25;
int obstacleX, obstacleY;


void setup(){
  size(300,200);
  posX = width/2;
  posY = height/2;
  velX = random(1,10);
  velY = random(1,10);
  fill(255,255,192);
  smooth();
  stroke(255,192,192);
}

void draw(){
  background(0);

  //actualitzamos las posiciones
  posX = posX+velX;
  posY = posY+velY;

  //dibujamos
  ellipse(posX,posY,sz,sz);

  //comprobamos la posiciÃ³n X
  if((posX<0)||(posX>width)){
    velX = -velX;
  }
  //comprobamos la posiciÃ³n Y
  if((posY<0)||(posY>height)){
    velY = -velY;
  }

  //actulizamos obstÃ¡culo, segÃºn el mouse:
  obstacleX = mouseX;
  obstacleY = mouseY;

  //dibujamos obstaculo:
  line(obstacleX,0,obstacleX,height);
  //COMPROBAMOS OBSTÃ CULO:
  if(abs(posX-obstacleX) < sz/2){
    velX = -velX;
  }
}
```

```
  //COMPROBAMOS OBSTÁCULO:
  if(abs(posX-obstacleX) < sz/2){
    velX = -velX;
  }
```

Generación de números aleatorios

Aunque por norma general no hay que abusar de ellos, los números aleatorios pueden ser muy útiles para dar un poco de "vida" a nuestros sketches. Por ejemplo, en el ejemplo de la pelota reboteadora (bouncing ball), siempre tenemos la misma velocidad X e Y, y esto produce al final que los movimientos sean siempre los mismos.Una manera de cambiar esto es darles a estos valores **velX** y **velY** un valor aleatorio.En Processing, para crear números aleatorios utilizamos la función **random()**. A **random()** le podemos enviar un o dos parámetros. Si le enviamos uno:

```
random(5);
```

nos devolverá un número **FLOAT** entre 0 y 5. Si le enviamos 2 nos devolverá un número **FLOAT** entre el primero y el segundo:

```
random(20,80);
```

¿Que significa "devuelve"? Lo veremos cuando toquemos funciones. Por ahora, hay que tener en cuenta que un **random()** lo asignaremos a un valor **float** así:

```
float miNumAleatorio = random(14);
```

Es muy importante tener en cuenta que **random()**, en Processing, devuelve siempre **floats**. Así pues, debemos o bien utilizarlo siempre con variables del tipo float(la solución fácil i recomendada), o sino deberemos **convertir** el valor que nos dé **random()** a un **int** (la solución pelín más compleja que por ahora no vale la pena). En todo caso esto se haría con la función de conversión **int()**.

Primero, un ejemplo simple de cómo utilizar un condicional junto con un **random()**. En cada iteración del proceso creamos un número aleatorio entre -1 y 1. Según sea mayor o menor a cero, vamos a hacer que este *viento* afecte a la dirección de la bola:

(Consultar código fuente: **vent).**

```
  //actualitzamos la posición según sople el viento:
  if(viento > 0){
    pos = pos+vel;
  }
  else { //oséase si el viento es menor o igual a 0:
    pos = pos-vel;
  }
```

He aquí un ejemplo de nuestra pelota reboteadora donde gracias a **random()**, en cada ejecución la pelota coge velocidades distintas. También puedes clicar para reiniciar la posición y velocidad de la bola. Mira **mousePressed** para ver cómo consultar código fuente: **bolaRev**

Estructuras de repetición

Las estructuras de repetición nos permiten ejecutar varias veces unas mismas líneas de código. En Processing, hay dos estructuras de repetición, de la que recomiendo utilizar casi exclusivamente la primera: **for** y **while**.

While repite una acción mientras (while) tal condición se cumple:

```
while(condición) {
 //instucciones;
}
```

Por ejemplo:

```
size(255,255);

int i=0;

while(i<height) {
    stroke(i,0,0);
    line(0, i, width, i);
    i = i + 2;
 }
```

Pero hay que ir con cuidado con este tipo de bucle: es muy fácil crear un bucle infinito, que en el caso de Processing hará que se cuelgue la aplicación. Esto pasaría si en el ejemplo anterior olvidamos la línia **i=i+2;** que es la que permite que en algún momento deje de cumplirse la condición **while(i<height)** y por consiguiente se rompa el bucle (y se continúe ejecutando el código).

2.- El "FOR LOOP"

Esta estructura de bucle es más utilizada que el *while*. De hecho funcionan de manera parecida, pero aquí la condición es sólo uno entre tres partes que controlan la estructura del bucle. Así, en el FOR tenemos una inicialización de una condición, su test, y una actualización. El bucle se ejecuta, pues, mientras el resultado del test sea *false*:

```
for(inicio; test; actualización) {
  //instucciones;
}
```

Una vez más, en abstracto esto suena extraterrestre, así que miraremos algunos ejemplos:

```
for(int i=0; i<3; i=i+1){
  println(i);
}
```

Analizemos este caso. Aquí, la inicialización es: **int i=0;**, es decir, declaramos una variable, de tipo **int** y nombre **i** de valor **cero**. Hasta aquí suena familiar. En segundo lugar tenemos el test: **i<10;.** Cuando éste proporcione como

resultado **true** se ejecutará el código que hay entre las claves del **bucle**, en este caso **println(i);** y seguidamente la actualización del bucle (la tercera parte de lo que hay entre paréntesis): **i=i+1** (**i** se incremente en **uno**). Así pues, en pseudocódigo, el ejemplo anterior se traduciría así:

```
for(int i=0; i<1000; i=i+1){
  println(i);
}
```

Pero vamos a lo gráfico. Imaginad que queremos crear un centenar de elipses en posición y de tamaño aleatorio. Sin un bucle, habría que hacer una misma operación cien veces. Pero con un **for loop** es tan fácil como hacer lo siguiente:

(Ver código fuente: **boles1).**

```
size(300,300);

 for(int i=0; i<100; i++){
     float posX = random(width);
     float posY = random(height);
     float tamano = random (100);
     ellipse(posX,posY,tamano,tamano);
}
```

Ejemplo que por cierto puede aplicarse sin problemas dentro del DRAW, como en este ejemplo (ver código fuente : **boles2)** donde puede comprobarse lo intenso que es el **smooth()** en según qué casos.

Un ejemplo donde el bucle nos sirve para realizar una acción un cierto número de veces, pero también podemos hacer que el mismo valor ("i" por convención) que incrementa el valor del bucle sea utilizado como un elemento gráfico más. Por ejemplo, consultad **boles3**

Vectores o arrays

Un Array es una lista de valores. Es un contenedor de múltiples variables a los que nos referimos por su posición dentro de dicha lista.

Una vez hemos visto bucles, condicionales, y variables, muy fácilmente uno se puede encontrar en una situación donde se empiezan a crear múltiples variables que tienen a ver entre ellas. Por ejemplo, si en nuestros ejemplos de la pelota queremos tener más de una, tenemos que hacer algo como **posX1**, **posX2**, **posX3**, etc. Podemos tener un sólo contenedor de posiciones: **posicionesX**, p.e., donde puede haber los valores que haga falta.

La cosa va así:

```
//creamos un array de enteros llamado listaPosiciones:
int[] listaPosiciones = { 14, 94, 120, 80 };
```

Vamos por partes: **int[]** indica que declaramos un array del tipo de datos **int**, nada más. Es decir, una lista de enteros. "**listaPosiciones**" es el nombre que le damos al **array,** y los números que van entre claves y separados por comas son sus valores. Para acceder a uno de estos valores, si con una variable normal nos basta con su nombre, en un array necesitamos el nombre del array y el índice.

Atención: el primer elemento de un array está siempre en la posición cero!. Así

```
listaPosiciones[2]
```

se refiere al tercer valor en el array. En el ejemplo anterior, 120. Se puede comprobar con el código que sigue:

```
//creamos un array de enteros llamada listaPosiciones:
int[] listaPosiciones = { 14, 94, 120, 80 };
//e imprimimos un par de sus valores a la consola:
println(listaPosiciones[0]);
println(listaPosiciones[3]);
```

Así, un array no es más que un conjunto de variables, que podemos utilizar como tales:

```
//creamos un array de enteros llamada listaPosiciones:
int[] listaPosiciones = { 14, 94, 120, 80 };
size(200,200);
// y dibujamos cuatro de elipses según el array:
ellipse(listaPosiciones[0], 100,25,25);
ellipse(listaPosiciones[1], 100,25,25);
ellipse(listaPosiciones[2], 100,25,25);
ellipse(listaPosiciones[3], 100,25,25);
```

También es útil saber que para cambiar un valor en un array, se realiza una asignación como con cualquier variable:

```
int[] listaPosiciones = { 14, 94, 120, 80 };

// pasan cosas...
//y cambio:
listaPosiciones[0] = 99;

println(listaPosiciones[0]);
```

En la sección de ejemplos se puede ver otra manera de declarar un array, que consiste en especificar el tamaño del mismo pero no el valor de lo que contiene:

```
//para un array de ints con 13 valors:
int[] unArray = new int[13];
```

Luego, a cada una de las posiciones del array que hemos creado, podemos asignar un valor tal y como lo hacemos con las variables normales.

```
unArray[0] = 18;
unArray[2] = int(random(80));
```

Para arrays grandes, este sistema mucho más eficiente en la mayoría de casos.

2.- Arrays y loops

Donde el poder de los **arrays** se aprovecha mejor, es en su combinación con los **bucles**. Es bastante lógico que si tenemos una estructura que puede guardar varias variables indexadas, y otra que nos permite iterar una misma acción en distintos elementos, las utilizaremos conjuntamente.

Si no, sólo hay que mirar el último ejemplo de la sección anterior, donde el hecho de estar utilizando un **array** no nos reduce para nada la cantidad de código que hay que escribir. Sería exactamente lo mismo si utilizáramos cuatro variables. En cambio, si combinamos el **array** con un **loop**, podemos hacer algo así:

```
int[] listaPosiciones = { 14, 94, 120, 80 };
size(200,200);
// y dibujamos las elipses via loop:
  for(int i=0; i<4; i++){
        ellipse(listaPosiciones[i], 100,25,25);
 }
```

El resultado es exactamente el mismo, pero aquí, gracias a utilizar **listaPosiciones[i]** dentro del **loop**, estamos aprovechando las iteraciones del bucle para afectar todos los elementos del **array**. Cuatro en este caso, pero que pueden ser muchos más.

```
int[] listaPosiciones = { 14, 25, 39, 64, 94, 109, 122, 150, 170, 178,
190 };
size(200,200);
// y dibujamos las elipses via loop:
  for(int i=0; i<11; i++){
        ellipse(listaPosiciones[i], 100,25,25);
  }
```

¿Pero qué pasa cuando tenemos ya muchos elementos? Contarlos puede llegar a ser un poco tedioso, y, sobretodo, ineficiente si cambiamos en algún momento la cantidad de valores que le damos al **array** en la declaración. Así, podemos substituir en el ejemplo anterior la línea de declaración de **for loop** por:

```
for(int i=0; i<listaPosiciones.length; i++){
```

Así utilizamos una variable que Processing ya calcula para nosotros: **array.length** que, como su nombre indica, corresponde a la cantidad de elementos presentes en el **array** en cuestión.

Finalmente, aprovechando el random para ser más eficientes, podemos hacer lo que sigue:

```
float[] listaPosiciones = new float[13];

size(200,200);
// creamos aleatoriamente todas las posiciones:
  for(int i=0; i<13; i++){
        listaPosiciones[i] = random(0,200);
  }
// y dibujamos las elipses via loop:
  for(int i=0; i<13; i++){
        ellipse(listaPosiciones[i], 100,25,25);
  }
```

3.- Ejemplos

Vimos en el punto uno que se pueden declarar variables sin asignarle valores inmediatamente.

Si utilizamos este código dentro de SETUP:

```
for(int i = 0; i<numeroBolas; i++){
    posicionesX[i] = width/2;
    posicionesY[i] = height/2;
    velocidadesX[i] = random(2,6);
    velocidadesY[i] = random(2,6);
}
```

Iniciamos una serie de posiciones en el eje X e Y en el punto medio del applet, y unas velocidades X e Y entre 2 y 6. Los cuatro **arrays** con los que trabajamos aquí son de **floats**.

Una vez hecho esto, sin cambiar casi nada el código del ejemplo con una sola pelota, podemos multiplicar los elementos que afectamos. Así, dentro del DRAW:

```
//iniciamos un bucle para que realize la acción para todos los
//elementos del array:
  for(int i = 0; i<numeroBolas; i++){
  //actualitzamos las posiciones
      posicionesX[i] += velocidadesX[i];
      posicionesY[i] += velocidadesY[i];
      //comprobamos los bordes X
```

```
        if((posicionesX[i]<0)||(posicionesX[i]>width)){
          velocidadesX[i] = -velocidadesX[i];
        }
      //comprobamos los bordes Y
        if((posicionesY[i]<0)||(posicionesY[i]>height)){
          velocidadesY[i] = -velocidadesY[i];
        }
  }
  //acabado el proceso, creamos otro bucle
  //donde dibujamos las elipses:
  for(int i = 0; i<numeroBolas; i++){
    ellipse(posicionesX[i],posicionesY[i],sz,sz);
  }
```

El ejemplo completo de las multibolas está aquí: **arraybball**

También podemos "resetear" utilizando una **función de sistema**:
el **mousePressed**:

```
void mousePressed(){
  //reinicializamos las velocidades:
  for(int i = 0; i<numeroBolas; i++){
    velocidadesX[i] = random(2,10);
    velocidadesY[i] = random(2,10);
  }
}
```

MousePressed es una función que hay que colocar **fuera del setup y del draw**, ya que se ejecuta independientemente de estos procesos.

Podéis mirar **colorMode()** para entender cómo se utiliza en el ejemplo el color, y el ejemplo de las bolas con colores: **arraybballcolors**

Otro ejemplo, un poco más complejo, pero que sirve para introducir el uso del texto puede revisarse en: **texttext**

Funciones

Las funciones nos sirven para organizar el código y poderlo reciclar luego. Una función es una estructura que en un momento dado en el flujo del programa es invocada. Cuando esto pasa, las instrucciones que estén contenidas en la función se van a ejecutar.

Entre otras ventajas, la funciones nos permiten organizar mejor el código Así, podemos por ejemplo crear un dibujo que requiere mucho código con una sola línea en el draw, y todo el código del dibujo contenido en una sola función, de manera que el DRAW nos queda así:

```
void draw(){
  background(255,127,127);
  drawTheBitxo();
}
```

(ver código fuente: **bitxo),**

Aquí, **drawTheBitxo();** invoca una función que hemos creado para dibujar el animal. Cuando llamamos la función, lo hacemos por su nombre, seguido de paréntesis. En el caso de una función que no recibe parámetros (como este), simplemente abrimos y cerramos paréntesis sin más para indicar que nos estamos refiriendo a una función. Para crear la función, hay que tener en cuenta dos cosas. Si hay **retorno** (si devuelve o no algún valor), y si tiene **parámetros**. Como dedicamos a éstas dos cosas sendos apartados en esta sesión, de momento nos centraremos en funciones que **no retornan nada** y **no reciben parámetros**.

También sería el caso siguiente, donde utilizamos una función para hacer otro dibujo:

```
void setup(){
  size(200,200);
}
void draw(){
  background(0);

  //invocamos la función, que definimos más adelante:
  dibujaUnSputnik();
}
//He aquí una función que dibuja un SPUTNIK:
void dibujaUnSputnik(){
  strokeWeight(3);
  stroke(255);
  fill(255,0,0);
  line(70,70,130,130);
  line(70,130,130,70);
  ellipse(100,100,35,35);
}
```

Aquí podemos ver cómo declaramos una función:

```
void dibujaUnSputnik(){
// instrucciones que forman la función
}
```

El **void** es la palabra clave que indica que la función no retorna nada. De la misma manera, los paréntesis en blanco indican que la función no recibe parámetros. Lo que estas dos cosas significan es que una función así se invocará de la siguiente manera:

```
dibujaUnSputnik();
```

Simplemente llamando la función por su nombre y los paréntesis en blanco provocaremos que en el punto en el código donde la función es invocada, el flujo de acciones salta hacia la llave donde empiezan las instrucciones de la función, sigue hasta que la función se cierra, y continúa luego hacia la línea que sigue la invocación.

2.- Parámetros

Lo que hemos visto hasta ahora no pasa de ser una mejor organización del código. Las funciones empiezan a ser mucho más interesantes cuando empezamos a utilizar los parámetros. Por ejemplo, podemos tener una función que nos dibuje algo allí donde está el mouse. Consúltense los siguientes códigos:

sptnk01, sptnk02_clic, sptnk03_pos

Aquí la gracia está en que en los tres casos utilizamos exactamente la misma función:

```
void dibujaUnSputnik(int x_, int y_){
  strokeWeight(3);
  stroke(255);
  fill(255,0,0);
  line(x_-30,y_-30,x_+30,y_+30);
  line(x_-30,y_+30,x_+30,y_-30);
  ellipse(x_,y_,35,35);
}
```

es decir, la **reutilizamos.** La misma función exactamente en tres sketch distintos. La diferencia está en el momento en que la función es invocada. En el primer y segundo ejemplos, en el DRAW, y en el tercero, en la función de sistema: MOUSEPRESSED, que se ejecuta cuando se clica el mouse. Pero aquí la función tiene una característica distinta a las del punto anterior. La función *dibujaUnSputnik* **recibe parámetros.** Concretamente recibe dos. Dos enteros. De aquí que al declarar la función no dejemos el paréntesis en blanco:

```
void dibujaUnSputnik(int x_, int y_){
```

Lo que está dentro del paréntesis son pues los parámetros, que se reciben en forma de variable. Unas variables que hay que declarar como vemos, y que llamamos en este caso **x_** e **y_**.

Los valores que se asignarán a dichas variables serán los de los parámetros que especifiquemos al asignar la función. Así,

```
dibujaUnSputnik(87, 349){
```

Asignaría a **x_** el valor 87 y a **y_** el valor 349 en dicha función. En nuestros ejemplos, lo que se les asigna es otra variable, en este caso variable de sistema, pero que no deja de ser un entero: **mouseX** y **mouseY**.

3.- Retorno

Otra cosa que pueden hacer las funciones, y aquí, aunque puede parecer que la cosa se complica, es cuando las cosas se vuelven interesantes. Además de poder o no recibir parámetros, una función puede o no devolver (retornar) valores. Hasta ahora hemos visto funciones que nunca devuelven nada. Este tipo de funciones simplemente se invocan, con o sin parámetros, sin más.

¿Pero qué pasa si una función devuelve un valor? ¿A quién se lo devuelve y cómo lo gestionamos? Pues de hecho es bien sencillo. ***Cuando invocamos una función que devuelve un valor, hemos de asignar éste valor a una variable*** (o utilizarlo directamente). Es algo que de hecho hemos estado ya haciendo con random.

Por ejemplo, si tenemos una función llamada calcul que nos realiza los cálculos de hacer la media entre dos valores, la asignaríamos a una variable llamada resultado:

```
int resultado;
resultado = calcul (32,439);
```

Y la función en sí sería algo así:

```
int calcul (int a_, int b_){
  int aRetornar;
  aRetornar = (a_+b_)/2;
  return aRetornar;
}
```

Otro aspecto muy importante ahora es la **declaración** de la función. Y aquí por fin cobrará sentido el famoso **void:** ***Cuando declaramos una función que devuelve un valor, debemos especificar el tipo de datos que devuelve al declararla*****.**

Así, una función que devuelve un entero se declararía por ejemplo así:

```
int dameUnEnteroBonito(){
  //instrucciones
}
```

Y una que nos ha de devolver un float:

```
float dameUnFlotador(){
   //instrucciones
}
```

Y, finalmente, ***hay que tener* siempre*, en una función que retorna algo, una instrucción de retorno al final***. Dicha instrucción se llama **return** y va seguida de un valor que se retorna, valga la redundancia...

```
float dameUnFlotador(){
   //instrucciones donde flotador es una variable del tipo float
   return flotador;
}
```

Como siempre, mejor con ejemplos: Imaginaros que tenemos dos equipos de programadores. Unos tienen que hacer un dibujo según lo rápido que se mueva el mouse. Los otros, contar los píxels blancos. Pues con funciones cada equipo podría trabajar por separado y al final integrarlo todo en un solo sketch. Consultadlo en **countdraw2**

Y otro ejemplo, con envío de parámetros y retorno en una sola función. Podemos utilizar arrays, loops y funciones para hacer cosas como contar la posición media de un grupo de elementos, como indica el ejemplo: **mitja**

4.- Más ejemplos

Podemos mezclar códigos ya explicados en el ejemplo: **bitxorebota**

O finalmente, otro ejemplo del uso de funciones y de paso introduciendo otras ideas de programación más interactiva: **follow**