

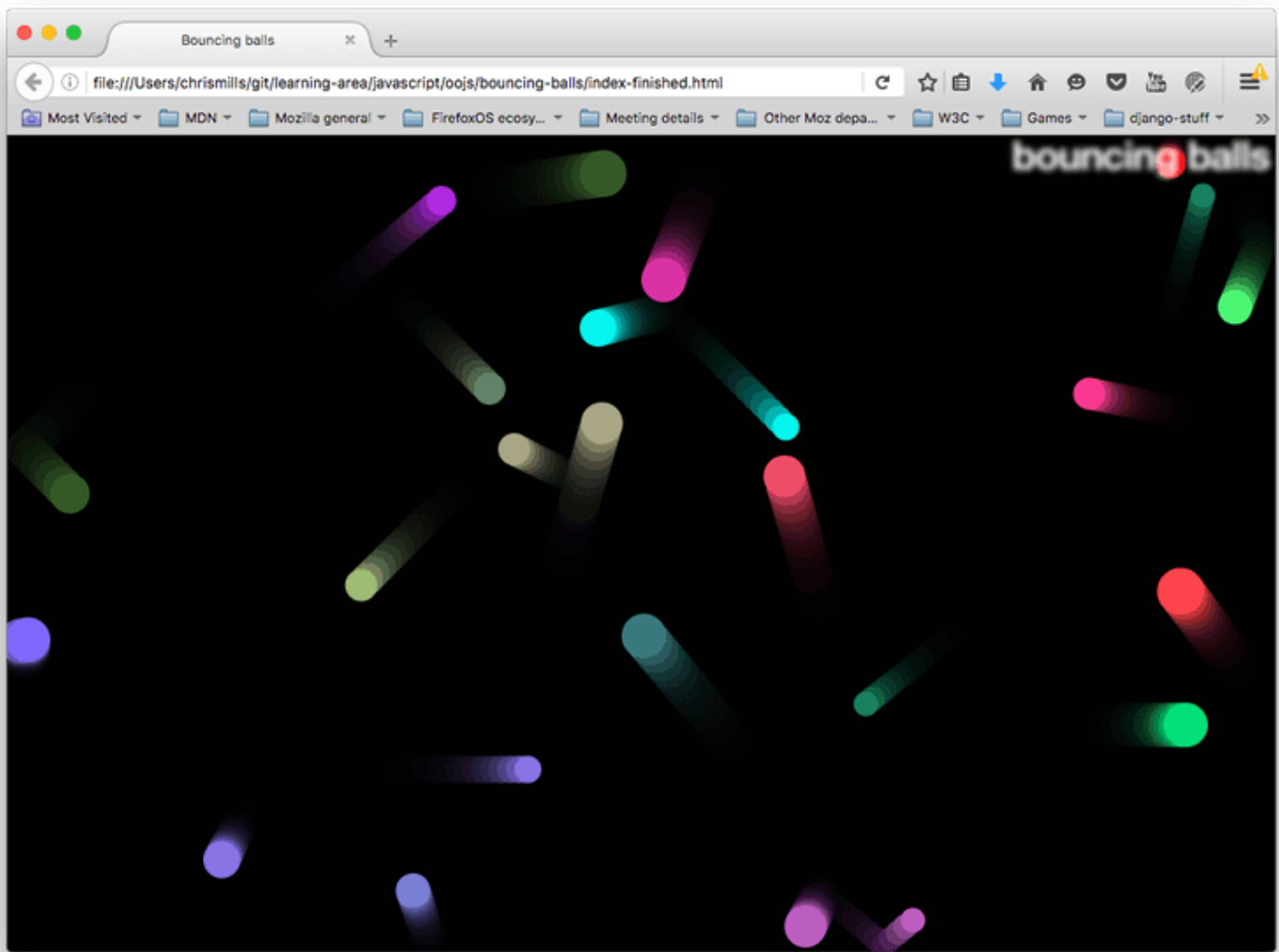
# Ejercicio práctico de construcción de objetos

En los artículos anteriores se explicó lo fundamental de la teoría de los objetos en JavaScript así como su sintaxis, para que Usted tenga un punto de partida sólido. En éste artículo, desarrollaremos un ejercicio práctico para ganar experiencia en la programación de objetos en JavaScript, con un resultado divertido y colorido.

<b>Pre-requisitos:</b>	Conocimientos básicos de computadores. Entendimiento básico de HTML y CSS. Familiaridad con los conceptos básicos de JavaScript (vea <a href="#">Primeros Pasos con JavaScript</a> y <a href="#">Elementos básicos de JavaScript</a> ) y OOJS (vea <a href="#">Conceptos básicos de los objetos JavaScript</a> ).
<b>Objetivos:</b>	Ganar experiencia en el uso de objetos y el uso de programación orientada a objetos en un contexto realista.

## Lanzemos algunas pelotas

Es éste artículo escribiremos un programa demo del juego clásico de pelotas que rebotan para mostrar la gran utilidad de los objetos en JavaScript. En éste demo las pelotas rebotaran en la pantalla y cambiaran de color cuando choquen unas con otras. Así, al final del ejemplo tendremos algo como esto:



En este ejemplo se utilizará [Canvas API](#) para dibujar las pelotas en la pantalla y la API [requestAnimationFrame](#) para animar todo el contenido de la pantalla. No es necesario que conozca estas funciones previamente. Esperamos que al final de este artículo, quizás pueda estar interesado en explorar su uso y capacidades más en detalle. Durante este desarrollo usaremos objetos y algunas técnicas para hacer que las pelotas puedan rebotar en los bordes y comprobar cuando choquen entre ellas (ésto se conoce como **detección de colisiones**).

## Primeros pasos

Para comenzar haga una copia en su computador de los archivos: [index.html](#), [style.css](#), y [main.js](#). Estos contienen:

1. Un documento HTML sencillo con un elemento `<h1>`, un elemento `<canvas>` en el que podamos dibujar los gráficos y otros elementos para aplicar los estilos CSS y el código JavaScript.
2. Algunos estilos sencillos que servirán para ubicar el elemento `<h1>`, ocultar la barra de desplazamiento y los márgenes del borde de la página (para que luzca mejor).

3. Un archivo JavaScript que sirve para definir el elemento `<canvas>` y las funciones que vamos a usar.

La primera parte del script es:

```
var canvas = document.querySelector('canvas');

var ctx = canvas.getContext('2d');

var width = canvas.width = window.innerWidth;
var height = canvas.height = window.innerHeight;
```



Este script obtiene una referencia del elemento `<canvas>`, luego llama al método `getContext()` para definir un contexto en el cual se pueda comenzar a dibujar. El resultado de la variable (`ctx`) es el objeto que representa directamente el área de dibujo del `<canvas>` y permite dibujar elementos 2D en él.

A continuación se da valor a las variables `width` and `height` que corresponden al ancho y alto del elemento *canvas* (representado por las propiedades `canvas.width` y `canvas.height`), de manera que el alto y ancho coincidan con el alto y ancho del navegador (*viewport*) cuyos valores se obtienen directamente de las propiedades `window.innerWidth` y `window.innerHeight`.

Puede ver que en el código se encadenan varias asignaciones, para obtener valores más rápidamente. Esto se puede hacer.

La última parte del script, es la siguiente:

```
function random(min, max) {
  var num = Math.floor(Math.random() * (max - min + 1)) + min;
  return num;
}
```



Esta función recibe dos números como argumentos de entrada (valor mínimo y máximo) y devuelve un número aleatorio entre ellos.

## Modelando una pelota en nuestro programa

Nuestro programa tendrá montones de pelotas rebotando por toda la pantalla. Ya que todas las pelotas tendrán el mismo comportamiento, tiene sentido representarlas con un objeto. Empezamos definiendo un constructor para el objeto pelota (*Ball*), en nuestro código.

```
function Ball(x, y, velX, velY, color, size) {  
  this.x = x; //posición horizontal  
  this.y = y; //posición vertical  
  this.velX = velX; //velocidad horizontal  
  this.velY = velY; //velocidad vertical  
  this.color = color; //color  
  this.size = size; //tamaño  
}
```



Aquí incluimos algunos parámetros que serán las propiedades que cada pelota necesita para funcionar en nuestro programa:

- las coordenadas `x` e `y` — correspondientes a la posición horizontal y vertical de la pelota. Estas pueden variar entre un valor 0 (en la esquina superior izquierda) hasta el valor del ancho y alto del navegador (esquina inferior derecha).
- velocidad horizontal y vertical (`velX` y `velY`) — cada pelota tiene una velocidad vertical y horizontal; en la parte práctica, estos valores se añadirán a las coordenadas `x` e `y` cuando animemos el movimiento de las pelotas, así en cada incremento de visualización de *frame*, se desplazarán esta cantidad.
- `color` — cada pelota posee un color.
- `size` — cada pelota tiene un tamaño, este será su radio en pixels.

Con esto se resuelven las propiedades del objeto, ¿Pero qué hacemos con los métodos? Ya que queremos que las pelotas realicen algo en nuestro programa.

## Dibujando las pelotas

Para dibujar, añadiremos el siguiente método `draw()` al prototipo del objeto `Ball()`:

```
Ball.prototype.draw = function() {  
  ctx.beginPath();  
  ctx.fillStyle = this.color;  
  ctx.arc(this.x, this.y, this.size, 0, 2 * Math.PI);  
  ctx.fill();  
}
```



Con esta función cada objeto pelota `Ball()` puede dibujarse en la pantalla utilizando el contexto 2D definido anteriormente (`ctx`)

- Primero usamos `beginPath()` para declarar que empezaremos a dibujar una forma en el *canvas*.

- A continuación usamos el `fillStyle` para definir el color de la forma. Haremos que coincida con la propiedad `color`.
- A continuación con el método `arc()` se traza un arco. Sus parámetros son:
  - La posición `x` e `y` del centro del arco. Corresponderán a las coordenadas del centro de la pelota.
  - El radio del arco - que vendrá dado por la propiedad de tamaño `size` de la pelota.
  - Los últimos dos parámetros especifican el comienzo y final del arco en radianes. En este caso se especifican `0` y `2*PI`. Que corresponden a `0` y `360` grados. Esto es un círculo completo. Si se quisiese especificar únicamente medio círculo, `180` grados, se especificaría `PI`.
- Por último con el método `fill()` se finaliza el dibujo, y rellena el área de la curva especificada, según se indico con el `fillStyle`.

Ya se puede empezar a testear el objeto.

1. Guarde el código hasta ahora, y cargue el archivo HTML en un navegador.
2. Abra la consola de JavaScript en el navegador, y refresque la página, para que el tamaño del *canvas* modifique sus dimensiones adaptándose al *viewport* con la consola abierta.
3. Teclee lo siguiente en la consola para crear una nueva pelota.

```
var testBall = new Ball(50, 100, 4, 4, 'blue', 10);
```



4. Pruebe a llamar a las variables miembro:

```
testBall.x  
testBall.size  
testBall.color  
testBall.draw()
```



5. Al teclear la última línea, debería ver que la pelota se dibuja en alguna parte del *canvas*.

## Actualizando los datos de la pelota

Ahora podemos dibujar una pelota en una posición dada, pero para empezar a moverla, se necesita una función de actualización de algún tipo. Podemos añadir el código a continuación, al final del archivo de JavaScript, para añadir un método de actualización `update()` en el prototipo de la clase `Ball()`

```
Ball.prototype.update = function() {  
  if ((this.x + this.size) >= width) {  
    this.velX = -(this.velX);  
  }  
}
```



```

if ((this.x - this.size) <= 0) {
    this.velX = -(this.velX);
}

if ((this.y + this.size) >= height) {
    this.velY = -(this.velY);
}

if ((this.y - this.size) <= 0) {
    this.velY = -(this.velY);
}

this.x += this.velX;
this.y += this.velY;
}

```

Las cuatro primeras partes de la función verifican si la pelota a alcanzado el borde del *canvas*. Si es así, se invierte la dirección de la velocidad, para que la pelota se mueva en la dirección contraria. Así, si la pelota va hacia arriba, (*velY* positiva) , entonces la velocidad vertical es cambiada, para que se mueva hacia abajo (*velY* negativa).

Los cuatro posibles casos son:

- Verificar si la coordenada *x* es mayor que el ancho del *canvas* (la pelota está saliendo por el borde derecho).
- Verificar si la coordenada *x* es menor que la coordenada 0 (la pelota está saliendo por el borde izquierdo)
- Verificar si la coordenada *y* es mayor que la altura del *canvas* (la pelota está saliendo por el borde inferior).
- Verificar si la coordenada *y* es menor que la coordenada 0 ( la pelota está saliendo por el borde superior).

En cada caso, se ha tenido en cuenta el tamaño (*size*) de la pelota en los cálculos, ya que las coordenadas *x* e *y* corresponden al centro de la pelota, pero lo que queremos ver es el borde de la pelota cuando choca con el perímetro del *canvas* — que la pelota rebote, cuando está a medio camino fuera de el —.

Las dos últimas líneas de código, suman la velocidad en x (*velX*) al valor de la coordenada *x* , y el valor de la velocidad en y (*velY*) a la coordenada *y* — con esto se consigue el efecto de que la pelota se mueva cada vez que este método es llamado.

Llegados a este punto: ¡continuemos, con las animaciones!

## Animando las pelotas

Hagamos esto divertido! Ahora vamos a empezar a añadir pelotas al canvas, y animándolas.

1. Primero, necesitamos algún sitio donde guardas las pelotas. El siguiente arreglo hará esta función — añádela al final de tu código.

```
;[] = var balls;
```

Todos los programas que generan animaciones normalmente tienen un bucle de animación, que sirve para actualizar los datos del programa, para entonces generar la imagen correspondiente; esta es la estrategia básica para la mayor parte de juegos y programas similares.

2. Añadamos las siguientes instrucciones al final del código:

```
function loop() {
  ctx.fillStyle = 'rgba(0, 0, 0, 0.25)';
  ctx.fillRect(0, 0, width, height);

  while (balls.length < 25) {
    var size = random(10,20);
    var ball = new Ball(
      // la posición de las pelotas, se dibujará al menos siempre
      // como mínimo a un ancho de la pelota de distancia al borde del
      // canvas, para evitar errores en el dibujo
      random(0 + size,width - size),
      random(0 + size,height - size),
      random(-7,7),
      random(-7,7),
      'rgb(' + random(0,255) + ',' + random(0,255) + ',' + random(0,255) +')',
      size
    );
    balls.push(ball);
  }

  for (var i = 0; i < balls.length; i++) {
    balls[i].draw();
    balls[i].update();
  }
}
```

```
requestAnimationFrame(loop);  
}
```

Nuestra función de bucle: `loop()`, hace lo siguiente:

- Define el color de relleno del canvas como negro semi-transparente, entonces dibuja un rectángulo en todo el ancho y alto del canvas, usando `fillRect()`, (los cuatro parámetros definen las coordenadas de origen, el ancho y el alto del rectángulo). Esto es para cubrir el dibujo del instante anterior antes de actualizar el nuevo dibujo. Si no se realiza este paso, resultará en las imágenes se irán apilando y veremos una especie de serpientes según se mueven por el canvas en vez de las pelotas moviéndose! El color de relleno se define como semitransparente, `rgba(0, 0, 0, 0.25)`, lo que nos permite que podamos intuir algunos de los dibujos de instantes anteriores, con lo que podremos recrear un poco el efecto de estelas detrás de las pelotas, según se mueven. Pruebe a variar este número para ver como resulta el efecto.
- Se crea una nueva instancia de la pelota `Ball()` usando un número aleatorio mediante la función `random()`, entonces se añade este elemento al final del arreglo de las pelotas, `push()`, pero unicamente si el número de pelotas es menor que 25. Así cuando tengamos 25 pelotas en la pantalla, no crearemos nuevas pelotas. Pruebe a variar el número de pelotas en el código: `balls.length < 25`. Dependiendo de la capacidad de procesamiento del navegador, un número de pelotas muy alto podría ralentizar significativamente la animación. ¡asi que cuidado!
- Se recorre el bucle por todo el conjunto de pelotas `balls` y se ejecuta el método para dibujar, `draw()`, cada una de las pelotas, y actualizar sus datos, `update()`, en cada una de ellas, así se conservarán las nuevas posiciones y velocidades para el siguiente intervalo de animación.
- Se ejecuta la función de nuevo mediante el método `requestAnimationFrame()` - cuando este método está continuamente ejecutándose y llama a la misma función, esto ejecutará la función de animación un determinado número de veces por segundo para crear una animación fluida. Esto se realiza normalmente de forma recursiva — lo que quiere decir que la función se llama a sí misma cada vez que se ejecuta, de esa manera se ejecutará una y otra vez de forma continua.

3. Por último, pero no menos importante, añadimos la siguiente línea, al final del código.-- es necesario llamar a la función inicialmente para que la animación comience.

```
loop();
```



Eso es todo para la parte básica — pruebe a guardar el código y refrescar el navegador para comprobar si aparecen las pelotas rebotando!



# Añadiendo la detección de colisiones




Ahora, un poco de diversión, añadamos la detección de colisiones a nuestro código. Así las pelotas, sabrán cuando chocan unas contra otras.

1. El primer paso, será añadir el código a continuación a continuación de donde se definió el método `update()`. (en código de `Ball.prototype.update`)

```
Ball.prototype.collisionDetect = function() {  
  for (var j = 0; j < balls.length; j++) {  
    if (!(this === balls[j])) {  
      var dx = this.x - balls[j].x;  
      var dy = this.y - balls[j].y;  
      var distance = Math.sqrt(dx * dx + dy * dy);  
  
      if (distance < this.size + balls[j].size) {  
        balls[j].color = this.color = 'rgb(' + random(0, 255) + ',' + random(0, 255) + ',' + random(0, 255) + ')';  
      }  
    }  
  }  
}
```

Esta función es un poco complicada, así que no hay que preocuparse mucho si de momento no se comprende del todo.

- Para cada pelota, necesitamos comprobar si chocará con cada una de las otras pelotas. Para esto, en un bucle `for` para recorrer todas las pelotas.
- Dentro del bucle, usamos un `if` para comprobar si la pelota que estamos mirando en ese ciclo del bucle `for` es la pelota que estamos mirando. No queremos mirar si una pelota ha chocado consigo misma. Para esto miramos si la pelota actual (es decir la pelota que está invocando al método que resuelve la detección de colisiones) es la misma que la indicada por el bucle. Usamos un operador `!` para indicar una negación en la comparación, así que el código dentro de la condición solo se ejecuta si estamos mirando dos pelotas distintas.
- Usamos un algoritmo común para comprobar la colisión de los dos pelotas. Básicamente miramos si el área de dos círculos se superponen. Esto se explica mejor en el enlace [detección de colision 2D](#).
- En este caso, únicamente se define la propiedad de `color` para las dos pelotas, cambiándolas a un nuevo color aleatorio. Se podría haber hecho cosas más complicadas, como que las pelotas rebotasen una con la otra de forma realista, pero esto habría supuesto un



desarrollo más complejo. Para desarrollar esos efectos de simulación física, los desarrolladores tienden a usar librerías de física como [PhysicsJS](#) , [matter.js](#) , [Phaser](#) , etc.

2. También es necesario llamar este método en cada instante de la animación. `balls[i].update()`; en la línea:

```
balls[i].collisionDetect();
```



3. Guardar y refrescar la demo de nuevo y podrá ver como las pelotas cambian de color cuando chocan entre ellas.

**Nota:** Si tiene problemas para hacer funcionar este ejemplo, puede comparar su código JavaScript, con el código de la [version\\_final](#)  (y también ver como funciona al [ejecutarla](#) .

## Resumen

Esperamos que se haya divertido escribiendo su propio mundo de pelotas que chocan aleatoriamente, usando objetos y programación orientada a objetos. Esto debería haberle dado una práctica útil y haber sido un buen ejemplo.

## Lea también

- [Canvas tutorial](#) — una guía de principiante para usar el canvas 2D.
- [requestAnimationFrame\(\)](#)
- [2D detección de colisiones](#)
- [3D detección de colisiones](#)
- [2D juego de ruptura usando sólo JavaScript](#) — un gran tutorial para principiantes sobre como construir un juego 2D.
- [2D juego de ruptura usando Phaser](#) — explica los conceptos fundamentales para construir un juego 2D usando una librería de juegos de JavaScript.

## En este módulo

- [Conceptos básicos de los objetos JavaScript](#)
- [JavaScript orientado a objetos para principiantes](#)
- [Prototipos de objetos](#)
- [Herencia en JavaScript](#)
- [Trabajando con datos JSON](#)

- [Ejercicio práctico de construcción de objetos](#)
- [Añadiendo características a nuestra demo de bouncing balls](#)