Guía de uso de programación: logotipo para la revista en línea *Mosaic*

Universitat Oberta de Catalunya

Índice

Autoría: Marc Padró

PID_00267138

Primera edición: septiembre 2019
Autoría: Marc Padró

Avda. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

Guía de uso de programación: logotipo para la revista en línea *Mosaic*

1. Estructura del proyecto

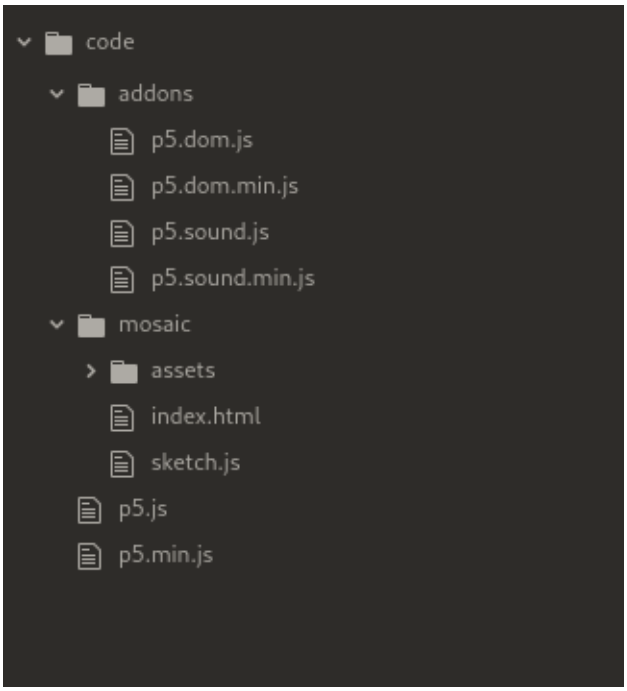
Un proyecto de p5.js, denominado *sketch*, consta principalmente de los ficheros siguientes:

- **index.html:** Contiene la estructura básica para acceder a nuestro *sketch* mediante un navegador web. En este documento, se enlazan el resto de ficheros del proyecto; lo tendremos que modificar si cambiamos la disposición o el nombre de alguno de estos ficheros o si añadimos otros nuevos, pero aparte de esto no hará falta que lo editemos. Podemos visualizar el resultado del proyecto abriendo este archivo con nuestro navegador.

- **p5.min.js o p5.js:** Contiene el código base de p5.js que necesitaremos para hacer funcionar cualquier *sketch*. El lenguaje de base del *sketch* es JavaScript, pero necesitamos este archivo para añadir toda la funcionalidad extra que nos permite escribir código de p5.js. Es lo que se denomina una *librería*.

- **sketch.js:** Es donde escribimos nuestro código.

Figura 1. Archivos y estructura del proyecto



Además, en este caso, en el directorio 'assets' se encuentran todos los recursos que cargamos en nuestro programa:

- **Imágenes para los números**, de 0 a 9. Para cada número hay una versión normal y una pequeña. Como son imágenes pequeñas y solo tienen un color y el fondo transparente, las hemos guardado en formato GIF, que es el que pesaba menos. Hemos usado el color blanco, porque de este modo después las podemos mostrar con el color que queramos usando la función 'tint'.

- **Imagen de fondo**. Usamos una imagen para el fondo, puesto que no tenerlo que dibujar desde el código nos ahorra mucho trabajo. La imagen tiene el doble de resolución de la medida en la que se mostrará para que se vea mejor en pantallas de alta densidad (*retina* y equivalentes). No usamos un SVG porque p5.js lo leería como un mapa de bits igualmente, y puede ser que en algunos casos no acabe de funcionar correctamente.

- **Tipografía** que usaremos para el texto, en formato TTF. A pesar de que en la versión definitiva usaremos una imagen, aquí utilizaremos texto para que sea más flexible. Dado que no podemos incluir la tipografía original, hemos usado una alternativa.

Figura 2. Contenido del directorio 'assets'

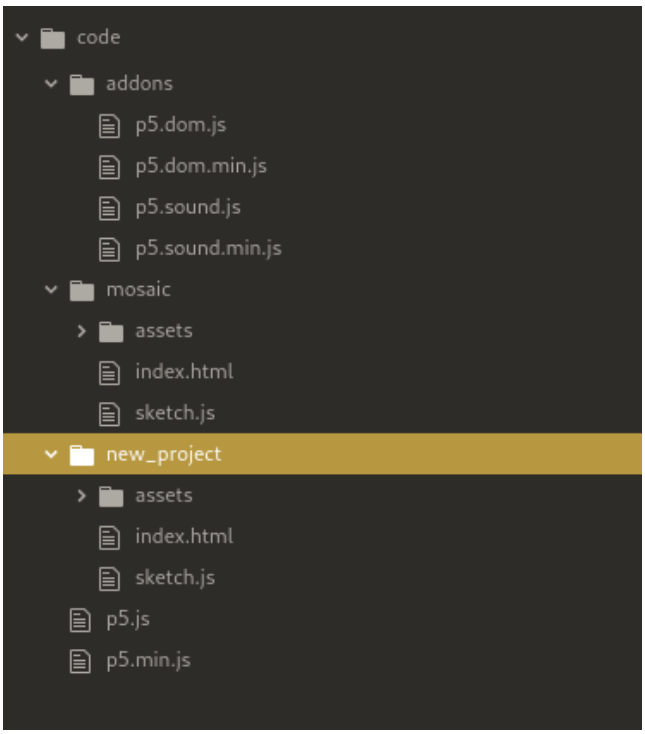
Dado que el objetivo será integrar el *sketch* en una plataforma WordPress mediante la extensión 'p5.js Embed' (podéis consultar el funcionamiento en la guía de uso correspondiente), al final solo necesitaremos el contenido del archivo 'sketch.js' y el directorio 'assets'. Pero mientras no hagamos esta integración podemos trabajar en nuestro *sketch* de manera aislada; entonces necesitaremos también el resto de ficheros.

2. Cómo trabajar en nuestro propio proyecto

2.1. Crear el proyecto

Para empezar a trabajar en vuestro propio proyecto, solo hace falta crear una nueva carpeta dentro del directorio 'code' y copiar los contenidos de la carpeta 'mosaic'. De este modo, podemos compartir la librería de p5.js entre todos los proyectos que creamos sin tenerla que duplicar. Si queréis empezar desde cero, podéis eliminar el contenido del archivo 'sketch.js' y todos los documentos del directorio 'assets'.

Figura 3. Podemos crear un nuevo proyecto copiando los contenidos de la carpeta 'mosaic'



Si queremos usar algún *addon* de p5.js (http://p5js.org/libraries/) necesitaremos enlazar el archivo correspondiente desde 'index.html'. Los archivos para las librerías p5.dom y p5.sound se encuentran ya en el directorio 'addons', pero tendremos que modificar ligeramente el archivo 'index.html' para que cargue las librerías que queramos.

Si abrís 'index.html' en un editor, veréis que hay dos líneas comentadas, es decir, entre los caracteres «<!--» y «-->».

Figura 4. Las líneas 9 y 10 del archivo index.html están comentadas

```
7   <style> body {padding: 0; margin: 0;} </style>
8   <script src="../p5.min.js"></script>
9   <!-- <script src="../addons/p5.dom.min.js"></script> -->
10  <!-- <script src="../addons/p5.sound.min.js"></script> -->
11  <script src="sketch.js"></script>
12 </head>
13 <body>
```

Podéis eliminar estos caracteres para enlazar la librería correspondiente.

Figura 5. Eliminamos los marcadores de comentario de la línea 10 para enlazar la librería p5.sound

```
7   <style> body {padding: 0; margin: 0;} </style>
8   <script src="../p5.min.js"></script>
9   <!-- <script src="../addons/p5.dom.min.js"></script> -->
10  <script src="../addons/p5.sound.min.js"></script>
11  <script src="sketch.js"></script>
12 </head>
13 <body>
```

Posiblemente os habéis dado cuenta de que tanto la librería principal como los 'addons' se enlazan mediante un archivo cuyo nombre contiene el texto «[...].min[...]»; pero que para cada librería hay también un archivo sin estos caracteres. Los archivos que contienen «min» en su nombre son archivos «minificados»; esto quiere decir que el código de estos archivos se ha modificado de forma automática para que ocupe el mínimo espacio posible, pero se ejecute del mismo modo. Así, los ficheros pesan menos y el navegador los puede cargar más rápidamente. La desventaja de usar archivos «minificados» es que no se puede entender el código, y en caso de producirse un error, puede ser más difícil conocer el motivo del mismo.

En este caso, los archivos sin minificar no son necesarios para el funcionamiento de nuestros programas, pero los hemos incluido por si alguien quiere ver cómo están escritas estas librerías. De todos modos, tampoco son los archivos originales en los que se programa la librería p5.js, sino que se hace en diferentes archivos que separan las funcionalidades de la librería por módulos. Podéis consultar los archivos originales de desarrollo en este enlace: https://github.com/processing/p5.js/tree/master/src.

2.2. Separar el código

Generalmente, si tenemos que trabajar en un proyecto complejo, nos interesa separar el código en varios documentos para tenerlo más estructurado y ordenado. En este ejemplo, sin embargo, lo hemos mantenido todo junto, puesto que a la hora de integrarlo en WordPress usando la extensión 'p5.js Embed' necesitamos tenerlo todo junto igualmente.

De todos modos, hemos separado las diferentes secciones que habríamos puesto en ficheros diferentes mediante el siguiente estilo de comentario:

```
/*
=====================================================================
    NOMBRE SECCIÓN
=====================================================================
*/
```

Encontraréis secciones principales para los ámbitos siguientes:

- Variables de configuración
- Programa principal
- Funciones de dibujo
- Clase 'Sector' (Utilidad para definir los cuatro sectores del logotipo)
- Clase 'Timer' (Utilidad para definir contadores de tiempo)

Si queréis separar vuestro código en diferentes ficheros, solo hace falta crear ficheros JS adicionales al mismo directorio que 'sketch.js' y enlazarlos desde 'index.html'. Para hacerlo, podéis duplicar la línea siguiente:

```
<script src="sketch.js"></script>
```

Y sustituir 'sketch.js' por el nombre del nuevo documento. Por ejemplo:

```
<script src="sketch.js"></script>
<script src="utils.js"></script>
```

2.3. El editor de código

Mientras que Processing incorpora su propio editor de código, p5.js no. Así pues, necesitaremos un editor.

2.3.1. Editor local

Un editor de código es un editor de texto que incorpora funcionalidades específicas que nos hacen más fácil programar, como por ejemplo, el coloreado automático de diferentes partes del código. Así pues, si bien podríamos editar los archivos de nuestro programa con un editor de texto plano, es prácticamente imprescindible usar un editor específico para código.

Estas son algunas opciones gratuitas y de código libre:

- Brackets (http://brackets.io/): Windows, OSX, Linux.
- Atom (https://atom.io/): Windows, OSX, Linux.
- Notepad++ (https://notepad-plus-plus.org/): Windows. Más sencillo que los anteriores, pero con todo lo que necesitáis.

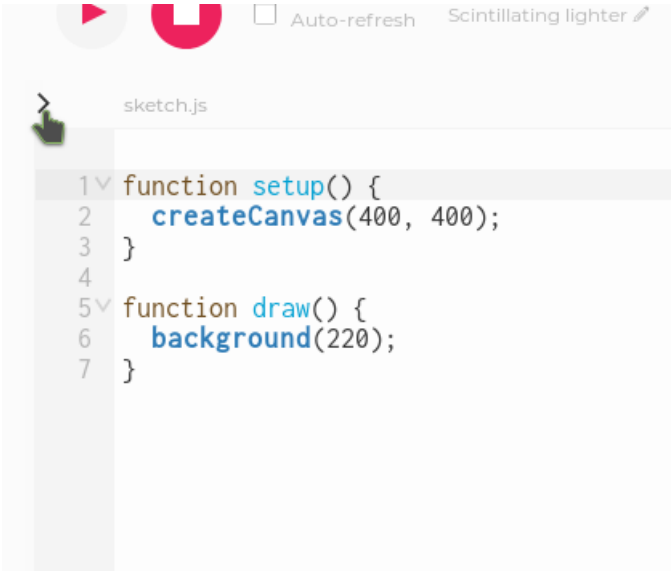
Cuando trabajamos de este modo, para ver el resultado de nuestro programa tendremos que abrir el fichero 'index.html' con el navegador web y volver a cargar la página cada vez que modificamos algún archivo.

2.3.2. El editor online de p5.js

Otra opción es usar el editor online de p5.js (https://editor.p5js.org/). Nos proporciona una experiencia más similar al editor de Processing y es muy cómodo si queremos empezar a hacer pruebas sin tener que preparar un proyecto nuevo. Si nos registramos, podemos guardar nuestros *sketchs* y acceder desde cualquier lugar. Por otro lado, si tenemos que añadir recursos a nuestro proyecto (imágenes, tipografías, etc.), no es tan cómodo como si tenemos los archivos en nuestro propio ordenador, puesto que los tenemos que subir cada vez que los modificamos y la interfaz no es especialmente fluida.

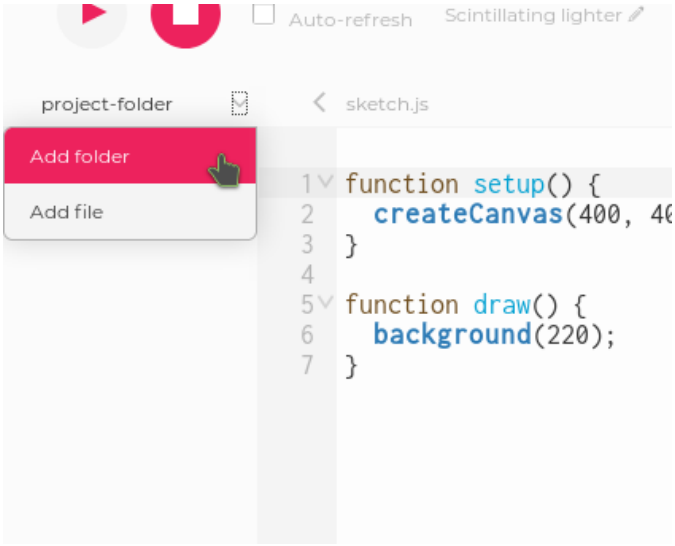
Si aun así queremos subir archivos al editor en línea de p5.js, antes que nada tendremos que estar registrados. Entonces tendremos que hacer clic en la flecha que encontramos sobre el espacio para el texto, para desplegar el inspector de ficheros.

Figura 6. Flecha para desplegar el inspector de ficheros en el editor online de p5.js



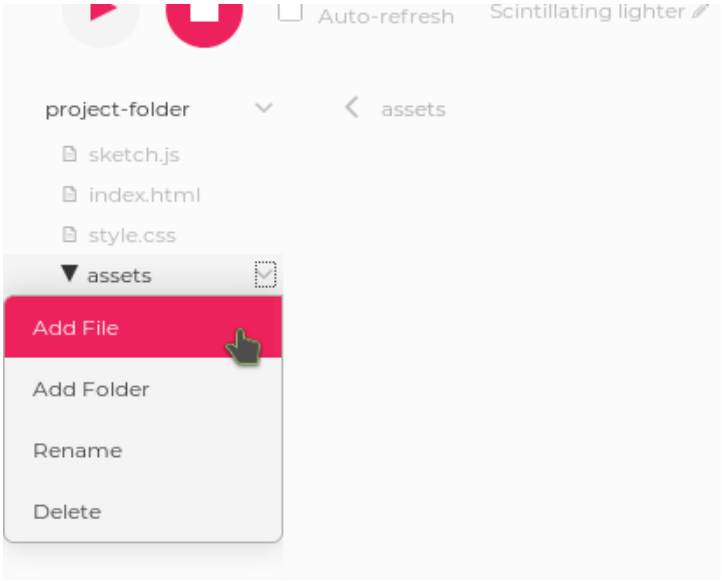
A continuación, si clicamos en la flecha junto a 'project-folder' se nos abre un menú que nos permite crear una carpeta o añadir ficheros. Creamos una carpeta 'assets'.

Figura 7. Menú para añadir carpetas y ficheros al proyecto del editor online de p5.js



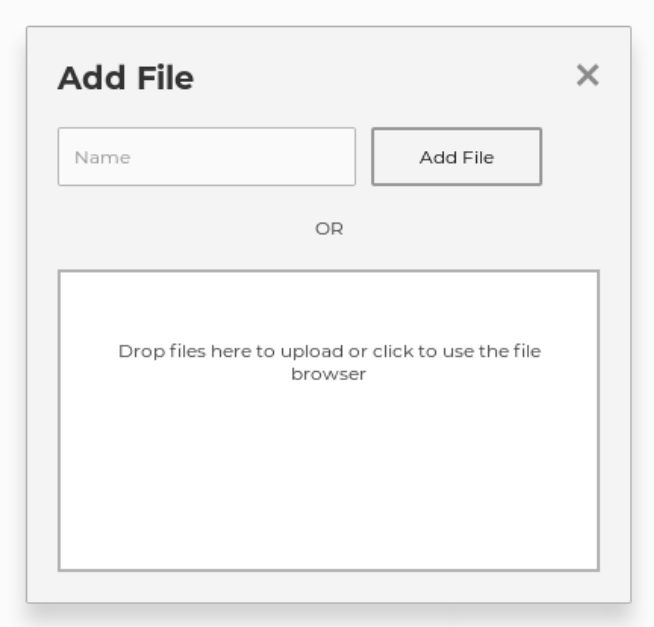
Ahora seleccionamos la carpeta que acabamos de crear y hacemos clic en la flecha que está a su derecha. Seleccionamos 'Add File'.

Figura 8. Añadimos archivos en la carpeta que hemos creado en el editor online de p5.js



En el diálogo que nos aparece, hacemos clic en el recuadro inferior para seleccionar los archivos que queremos subir o los arrastramos y los soltamos encima.

Figura 9. Diálogo para añadir ficheros en el editor online de p5.js



Una vez se han acabado de subir los ficheros, podemos cerrar el diálogo y volver a seleccionar ‘sketch.js’ para seguir editándolo.

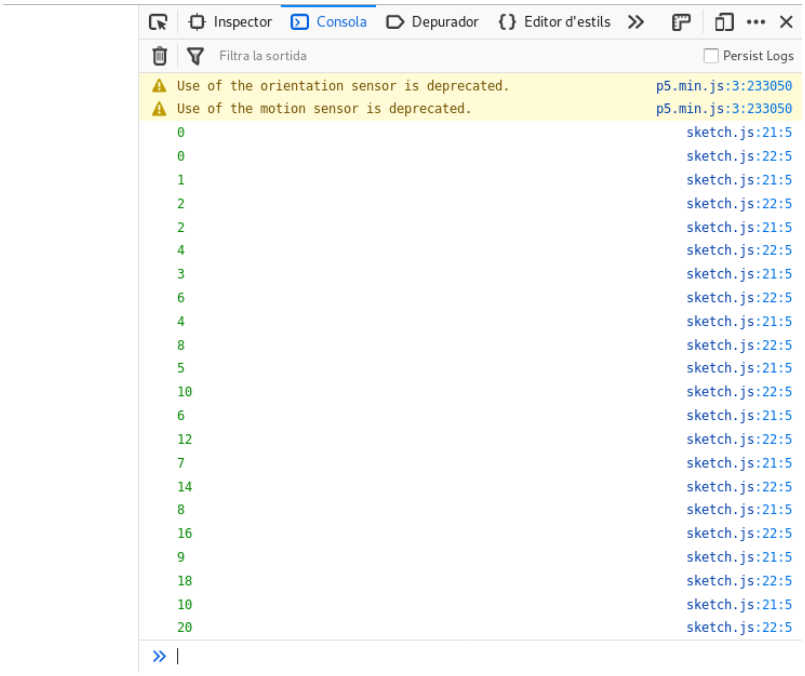
3. La consola de JavaScript

Mientras estamos desarrollando nuestro código, nos puede resultar útil mostrar el valor de determinadas variables. Para hacerlo podemos usar la consola de JavaScript.

La consola de JavaScript es una utilidad de los navegadores web. Según nuestro navegador la podemos abrir de la manera siguiente:

- **Firefox:** En ‘Herramientas’ > ‘Desarrollador web’ > ‘Consola web’.
- **Chrome:** En el menú > ‘Más herramientas’ > ‘Herramientas para desarrolladores’, y seleccionamos la pestaña ‘Console’.
- **Edge:** Pulsamos la tecla F12 y seleccionamos la pestaña ‘Console’.
- **Safari:** Primero tenemos que tener activadas las opciones de desarrollador. Para hacerlo, vamos a ‘Safari’ > ‘Preferencias’, y en la pestaña ‘Avanzado’ marcamos la opción «Muestra las opciones de desarrollador». Una vez hecho esto, podemos ir al menú ‘Desarrollador’ > ‘Muestra la consola de JavaScript ’.

Figura 10. Ejemplo de consola de JavaScript



Para mostrar variables en la consola, p5.js nos facilita la función 'print'. Nosotros recomendamos usar 'console.log' en lugar de 'print', puesto que es equivalente y es la manera propia de JavaScript de hacerlo.

```
function setup() {
	var texto = "Hello World!";
	print( texto ); // Muestra "Hello World!" en la consola
	console.log( texto ); // También muestra "Hello World!" en la
consola
}
```

Además, la función 'print' no funciona igual fuera de las funciones propias de p5.js ('preload', 'setup', 'draw', etc.), puesto que sirve para abrir el diálogo de la impresora. 'console.log', en cambio, se puede usar en todas partes:

```
var texto = "Hello World!";
print( texto ); // Abre el diálogo de impresión
console.log( texto ); // Muestra "Hello World!" en la consola
function setup() {
	print( texto ); // Aquí sí, muestra "Hello World!" en la consola
}
```

Veréis que en la consola a veces también aparecen otros mensajes, como por ejemplo, errores y avisos del navegador. Algunos avisos, como los que se muestran en la figura 10, puede ser que os aparezcan siempre. No hace falta que hagáis caso de ellos. Pero si vuestro código no funciona, puede ser que en la consola se muestre algún error que os ayude a saber qué ha pasado.

Si usáis el editor en línea de p5.js, veréis que ya lleva una consola incorporada. Por lo tanto, no es necesario abrir la consola del navegador. De todos modos, a veces la información que aparece es más completa y nos es más cómodo usarla.

4. *let* y var

Cuando consultáis ejemplos de código de p5.js (y JavaScript en general), a veces encontraréis las variables declaradas con la palabra *let* y otras con la palabra *var*. Generalmente estas dos palabras son intercambiables, a pesar de que *let* forma parte de una sintaxis más reciente y puede ser que algunos navegadores más antiguos no la soporten.

La principal diferencia entre las dos está en el ámbito: la zona en la cual la variable existe una vez declarada. El ámbito de *var* son las funciones, es decir, que no podemos acceder a aquella variable desde fuera de la función donde la hemos declarado. El ámbito de *let* son los bloques. Un bloque es cualquier fragmento de código encapsulado entre { }. La declaración de una función también constituye un bloque, pero no es el único caso.

Por ejemplo, el código siguiente sería válido:

```
...
if ( foo > 0) {
     var texto = "Texto 1";
}
else {
     var texto = "Texto 2";
}
console.log( texto );
```

Mientras que, en este otro caso, la consola mostraría 'undefined' o nos daría un error:

```
...
if ( foo > 0) {
     let texto = "Texto 1";
}
else {
     let texto = "Texto 2";
}
console.log( texto );// texto no existe
```

Así pues, el ámbito de *let* es más restrictivo que el de *var*.

Declarar variables dentro de condicionales si las queremos usar fuera del condicional no es recomendable, sino que es más aconsejable declararlas antes y asignarles el valor dentro del condicional. De este modo, es más evidente que queremos usar aquella variable fuera del condicional y evitamos, por ejemplo, el descuido de eliminar la declaración de la variable pensándonos que no la usaremos más adelante. Puede ser que en proyectos sencillos esto no pase, pero cuando trabajamos en proyectos más complejos, donde quizás hacemos varias cosas dentro del condicional, estos descuidos son muy corrientes.

Por lo tanto, reescribiríamos el ejemplo anterior de la manera siguiente:

```
...
let texto;
if ( foo > 0) {
     texto= "Texto1";
}
else {
     texto= "Texto2";
}
console.log( texto);
```

Si usamos *let*, por lo tanto, nos obligamos a seguir buenas prácticas al escribir código, y así será más fácil de leer y de mantener.

Hay una tercera manera de declarar variables que es mediante la palabra *const*. Es equivalente a usar *let*, pero con la diferencia de que no se le puede reasignar un nuevo valor; la palabra *const* viene de 'constante'. No la hemos usado para no hacer el código más complejo, pero si os interesa, podéis buscar información, puesto que el uso mejora la legibilidad del código.

5. Objetos

El código del logotipo de *Mosaic* usa un uso extensivo de objetos. Los objetos nos permiten agrupar diferentes valores en una única variable. A diferencia de los 'arrays', estos valores se identifican con nombres en lugar de números. Es como tener variables dentro de una variable.

Cada uno de los valores de un objeto se denomina *propiedad*. JavaScript nos permite crear objetos genéricos a los cuales podemos asignar las propiedades que queramos.

Así pues, podemos crear el objeto siguiente:

```
let punto = {
        x: 1,
        y: 3,
        nombre: "A",
};
```

Como veis, podemos crear un objeto definiendo las propiedades entre claves ({ }). Para cada propiedad escribimos el nombre, dos puntos y el valor. Como las variables de JavaScript, las propiedades de los objetos pueden contener cualquier valor de cualquier tipo.

Entonces podemos acceder a las propiedades, modificarlas e incluso añadir de la manera siguiente:

```
...
console.log( punto.nombre ); // Muestra "A" en la consola
punto.y = 2; // Modificamos la propiedad 'y'
punto.z = -1; // Añadimos una propiedad 'z'
```

También podemos crear objetos vacíos a los cuales podemos añadir propiedades posteriormente:

```
let obj = {};
obj.nombre = "A";
```

Otra sintaxis para acceder a las propiedades de los objetos es la siguiente:

```
let persona = {
        nombre: "Mireia",
        edad: 32,
};
console.log( persona[ "nombre" ] );
persona[ "edad" ] += 1;
```

Fijaos que en este caso tenemos que escribir el nombre de la propiedad entre comillas. Sin embargo, también podemos guardar el nombre de la propiedad en una variable:

```
let persona = {
        nombre: "Mireia",
        edad: 32,
};
let propiedad = "nombre";
console.log( persona[ propiedad ] ); // Muestra "Mireia" en la consola
```

En este ejemplo tan sencillo esta sintaxis resulta demasiado complicada, pero nos es muy útil si queremos acceder a las propiedades de uno o varios objetos de manera programática.

6. *class*

En el código del logotipo de *Mosaic* os encontraréis con otra sintaxis con la cual quizás no estáis familiarizados: la declaración *class*.

Si habéis usado Processing, puede ser que os hayáis encontrado con su equivalente en el lenguaje Java. El funcionamiento es similar.

La declaración *class* nos permite crear un patrón para definir nuevos objetos. Si habéis leído el apartado «Objects» del libro *Make: Getting Started with p5.js*, debéis de haber visto que usan funciones para crear objetos que responden a un patrón. Con *class* conseguimos el mismo resultado y técnicamente es más adecuado. Además, es más fácil de leer.

Vemos un ejemplo:

```
class Counter {
	constructor( incr ) {
		this.val = 0;
		this.incr = incr;
	}
	tick() {
		 this.val += this.incr;
	}
}
let count1 = new Counter (1);
let count2 = new Counter (2);
function setup() {
	frameRate(2);
}
function draw() {
	console.log( count1.val );
	console.log( count2.val );
	count1.tick(); // Incrementamos la propiedad val de count1
	count2.tick(); // Incrementamos la propiedad val de count2
}
```

Si ejecutáis este código la consola irá mostrando los valores siguientes: 0, 0, 1, 2, 2, 4, 3, 6, 4, 8…

Tenemos dos contadores: uno que avanza uno por uno y el otro que avanza de dos en dos. Pero podríamos crear todos los que quisiéramos con los incrementos que quisiéramos.

Como podéis ver, la declaración *class* consta de un nombre de clase (en este caso 'Counter') y de una serie de métodos. Fijaos que escribimos la primera letra de los nombres de las clases en mayúscula, para diferenciarlas de los nombres de las variables.

Denominamos *métodos* las funciones que forman parte de un objeto o una clase. Pero cuando definimos una clase, no tenemos que escribir la palabra *function* ante los métodos.

El primer método de todos, 'constructor', es un método especial y es el principal de las clases, puesto que es el que se ejecuta cuando creamos una nueva instancia de la clase: cuando creamos un nuevo objeto. Para hacerlo, usamos el operador *new*:

```
...
let count1 = new Counter (1);
let count2 = new Counter (2);
...
```

Como podéis ver, pasamos en la clase los argumentos que hemos definido a 'constructor', puesto que lo que pasa en realidad cuando usamos el operador *new* es que se ejecuta el método 'constructor' y se retorna el objeto creado. Fijaos que dentro de 'constructor' usamos la variable 'this'.

```
...
        constructor( incr ) {
                this.val = 0;
                this.incr = incr;
        }
...
```

'this' hace referencia al objeto que estamos creando. Del mismo modo, en cualquiera de los otros métodos de la clase podemos hacer referencia a la variable 'this' para acceder a las propiedades y a los otros métodos de la instancia.

```
...
        tick() {
                 this.val += this.incr;
        }
...
```

Así, una vez creamos una instancia de la clase, podemos acceder a sus métodos, que solo afectarán a las propiedades de aquella instancia. Como veis, también podemos acceder directamente a sus propiedades:

```
...
        console.log( count1.val );
        console.log( count2.val );
        count1.tick(); // Incrementamos la propiedad val de count1
        count2.tick(); // Incrementamos la propiedad val de count2
...
```

La sintaxis de *class* es relativamente reciente y puede ser que no funcione en algunos navegadores más antiguos. Hay otra manera de conseguir el mismo resultado, los 'prototipos', que son la manera tradicional de hacerlo en JavaScript. El funcionamiento puede resultar más complicado de entender, a pesar de que en realidad *class* usa de manera interna los prototipos. Los prototipos son una faceta muy flexible de JavaScript de la cual podéis buscar más información si os interesa.

7. Integración con WordPress

Para integrar nuestro proyecto en un WordPress usaremos la extensión 'p5.js Embed'. Leed la guía de uso correspondiente para saber cómo usarla.

'p5.js Embed' nos proporciona una variable 'wp_data', donde encontramos información relativa al sitio web y a la página que se está consultando. Si trabajamos en nuestro *sketch* de manera aislada, no dispondremos de esta variable. Si queremos que nuestro código funcione tanto dentro como fuera de WordPress, podemos comprobar si la variable 'wp_data' existe, y, en caso de que no sea así, usar una serie de valores por defecto. Por ejemplo:

```
let post_count;

if ( typeof( wp_data ) !== "undefined" ) {
       post_count = wp_data.post_count;
}
else {
       post_count = 10;
}
```

'typeof' es una función de JavaScript que retorna el tipo de una variable. Cuando la variable no ha sido definida, retorna como «undefined».