UD4

Objetos definidos por el usuario. Funciones.

Objetos (sin Orientación a Objetos)

En unidades anteriores hemos estudiado una serie de objetos predefinidos por el sistema; entre ellos estaban los tipos primitivos los cuales solo podían ser utilizados para contener un valor de un solo tipo: un String almacena pues eso, una cadena, un Number contiene un número, ni más ni menos.

Los objetos que trataremos en esta unidad son aquellos que serán usados para almacenar una colección de elementos de igual o distinto tipo.

Generalmente hay 2 formas sencillas de crear un objeto:

1. Un **objeto** puede ser creado usando un constructor de la clase **Object():**

```
let coche = new Object();
coche.puertas = "2";
coche.ruedas = "4";
coche.color = "azul";
coche.darColor = function () { “Mi color es: ” + coche.color};
```

2. O bien con **2 llaves { }** y situando en su interior una **lista de pares “clave : valor”**. Veamos un ejemplo sencillo de cómo crear un objeto:

```
let coche = {
  puertas: "2",
  ruedas: "4",
  color: "azul",
  darColor : function() { return `Mi color es ${coche.color}`};
  darColor : function() { return `Mi color es ${this.color}`};
};
```

Aquí hemos creado 2 usuarios (a priori, objetos vacíos) utilizando distintos métodos de instanciación: usando su constructor por defecto, y directamente las 2 llaves. Ninguno de estos objetos tiene ninguna propiedad ni métodos.

Propiedades

En el siguiente ejemplo podemos ver como creamos un objeto Usuario con 2 propiedades:

```
let usuario = {
nombre : "Tomasa"   ,    // Propiedad de tipo String denominada "nombre".
edad : 17                // Propiedad de tipo Number denominada "edad"
};
```

Importante: Fíjate en la coma como separador de las propiedades del objeto.

También podríamos haber definido el objeto “usuario” haciendo uso del **operador “new”** de la siguiente forma:

```
let usuario = new Object();
usuario.nombre = "Tomasa";
usuario.edad = 17;
```

Acceso a las propiedades

Podemos acceder a las propiedades de un objeto de 3 formas principalmente:

1. Usando el operador “.”:

Como en muchos lenguajes de programación, podemos acceder a las propiedades de un objeto usando el **operador punto “.”** :

```
console.log( "Nombre del usuario:" + usuario.nombre );
console.log("Edad del usuario:" + usuario.edad );
```

2. Usando el nombre de la propiedad que queremos acceder:

Para esto usaremos los **corchetes [“nombre_propiedad”]:**

```
console.log("Nombre del usuario: " + usuario["nombre"]);
```

3. Usando una variable que contenga el valor de la propiedad que queremos acceder:

```
var nombrePropiedad = "nombre";
console.log("Nombre del usuario: " + usuario[nombrePropiedad]);

Otro ejemplo:

let usuario = { nombre = "Peter"  ,  apellido = "Pan"  , edad = 103 };
let clave = prompt ( "¿Qué valor desea conocer sobre su perfil: nombre, apellido o
edad ? ");
console.log ( usuario[clave] );
```

Asignar/Modificar valores a una propiedad

Lo mismo que antes, usaremos el operador punto para acceder a las propiedades y poder asignar o modificar su valor actual:

```
usuario.edad = 17.5;
```

Eliminar una propiedad

Javascript tiene *“la potencia”* de dejarnos eliminar propiedades, y de una forma muy sutil: haciendo uso de la **palabra “delete”:**

```
delete  usuario.edad;
```

Recorrer las variables de un objeto: **for.....in.....**

Para recorrer “todas” las variables de un objeto normalmente utilizaremos la estructura de control **“for...in...”**.

Esta estructura **no nos garantiza el recorrer las variables en un orden determinado**, sino que las recorrer tal como se han creado:

```
let usuario = { nombre : "Tomasa" , edad : 17  };

for ( propi in usuario)
{
    console.log("Propiedad= " + propi + "con valor " + usuario[propi] );
}
```

Funciones dentro de un objeto: método

Método es el nombre que recibe una función cuando forma parte de un objeto. Dado que en JavaScript las funciones pueden ser **almacenadas como variables**, el hecho de asignar una función a una propiedad la convierte en un método. Estos funcionan exactamente igual que una función, pero **su contexto de ejecución (this) es el objeto**.

La sintaxis para crear una función dentro de un objeto es la siguiente:

<nombre_método> : function (arg1, arg2, ...) { // instrucciones };

Y el acceso al mismo sería de la forma:

<nombre_objeto> . <nombre_método>();

Veamos un ejemplo:

```
let usuario = {
  nombre : "Tomasa" ,
  edad : 17 ,
  devolverDatosUsuario : function () { return "Nombre: " + this.nombre + " - Edad: "
+ this.edad };
}
usuario.devolverDatosUsuario();
```

También podríamos crearlo de la siguiente forma:

```
var objeto = new Object();
objeto.saludar = function() { console.log("Hola amigo!"); };
```

Método 1: Construyendo objetos mediante el Método Clásico.

Construir objetos mediante funciones constructoras

En JavaScript es muy común ver la creación de objetos haciendo uso de funciones. De esta forma podemos crear varios objetos del mismo tipo de una forma más sencilla.

```
function  Usuario ( nombre, edad)  {
      this.nombre = nombre;
      this.edad = edad;
}
let  usuario  =  new Usuario( "Sabonis" , 54 );
console.log ( usuario.nombre);   // Sabonis
console.log(  typeof Usuario);   // function
console.log(  typeof usuario);   // object
```

Prototipos

Todos los objetos tienen una propiedad especial llamada **“prototype”** que a su vez no es más que una referencia al propio objeto mismo el cual contendrá todas las propiedades y métodos del objeto. Con los prototipos se nos facilita muchísimo la creación de objetos de un determinado objeto, así como el modificarlos añadiendo nuevas propiedades o funciones.

```
/* Sintaxis de creación de un prototipo de un objeto usando la función constructor:
*/
       function Persona (nom, ape, an ){
             this.nombre = nom;
             this.apellido = ape;
             this.ano = an;
             this.nombreCompleto = function (){
                   return this.nombre + " " + this.apellido;
             }
       }
       var lina = new Persona ("Lina", "Morgan", "1815");
       var fernando = new Persona ("Fernando", "Esteso", "1791");
```

Ahora vamos a añadir una propiedad al objeto que acabamos de crear:

```
       /* Añadir una propiedad a un objeto */
       lina.nacionalidad = "Inglesa"; //Solo se añade a ada, no a charles
       alert( lina.nacionalidad);    //Devuelve "Inglesa"
       alert( fernando.nacionalidad); //Devuelve "Undefined"
```

Aquí lo que ha pasado es que hemos añadido una nueva propiedad al objeto “lina” pero este no ha sido añadido al objeto “fernando”. ¿Y si añadimos también el método “nacimiento” a lina?

```
/* Añadir un método a un objeto */
	lina.nacimiento = function(){
		return "Nacimiento en el año "+this.ano;
	} //Solo se añade a ada, no a charles
	alert(lina.nacimiento()); //Devuelve "Nacimiento en el año 1815"
	alert(fernando.nacimiento); //Devuelve "Undefined"
```

Aquí lo que necesitamos es hacer uso de la propiedad “prototype” en el caso de que quisiéramos crear una propiedad o una función nueva para todos los objetos que deriven del objeto en cuestión usaremos los prototipos:

- Añadir una **propiedad** a un prototipo: **<Nombre_objeto>.prototype.<propiedad>**

```
Persona.prototype.muerte = "2000";
```

- Añadir un **método** a un prototipo: **<Nombre_objeto>.prototype.<método> =...**

```
Persona.prototype.defuncion = function(){
		return "Defunción en el año "+this.muerte;
	}
var alan = new Persona("Alan", "Turing", "1912");
alert (alan.nombre);
alert (alan.defuncion());
```

Herencia mediante prototipos

Mediante los prototipos de javascript podemos crear objetos que hereden de otros objetos. Veamos con un ejemplo el proceso:

Primero crearemos el objeto Doméstico con su función constructora:

```
function Domestico() {
	this.animal ="";
	this.nombre="";
	this.configurarAnimal = function (nuevoAnimal) {
	   this.animal = nuevoAnimal;
	}
	this.configurarNombre = function (nuevoNombre){
	   this.nombre  = nuevoNombre;
	}
};
```

Probamos a crear un objeto de tipo Doméstico:

```
var miGato = new Domestico();
miGato.configurarAnimal("Gato");
miGato.configurarNombre("Filomeno");
console.log(miGato);
```

Ahora vamos a crear un **objeto Perro que hereda de doméstico**; para ello solo usaremos la

palabra **"prototype".**

```
Perro.prototype = new Domestico();  //Ya tenemos la herencia configurada
```

Y ahora vamos a darle forma a Perro con su método constructor ofreciendo nuevas funcionalidades:

```
function Perro () {
      this.raza = "";
      this.configurarRaza = function (razaNueva) {
            this.raza = razaNueva;
      }
      this.mostraInfo = function (){
            console.log ( "El perro con nombre " + this.nombre + " es de raza " +
this.raza);
      }
}
```

Y ahora vamos a probar que el proceso de herencia ha sido realizado satisfactoriamente:

```
var perrito = new Perro();
perrito.nombre = "Tobi";
perrito.configurarRaza("Fox Terrier");
perrito.mostraInfo();
```

Herencia mediante prototipos: construyendo parámetros del objeto padre

Hasta ahora los ejemplos que habíamos visto de herencia no hacían ningún tipo de construcción de los objetos (tanto hijo como parámetro) haciendo uso de sus parámetros del constructor. Para este nuevo caso vamos a utilizar una nueva instrucción en el objeto hijo para que pueda desde este construir al padre. Esta nueva instrucción será:

```
      Padre.prototype.constructor.call( this, parametroPadre1, parámetroPadre2);
```

Veámoslo con un ejemplo:

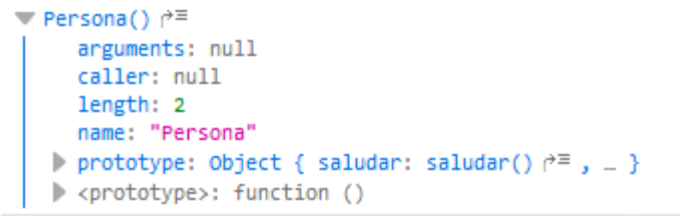
```
 function Persona (dni, nombre) {
            this.dni = dni;
            this.nombre = nombre;
      }
 Persona.prototype.saludar = function () {
      console.log("Hola me llamo " + this.nombre);
 }
      //Heredamos de Persona
      function Estudiante (dni, nombre, numeroMatricula) {
            Persona.prototype.constructor.call(this, dni, nombre);
            this.numeroMatricula = numeroMatricula;
      }
```

```
Estudiante.prototype = new Persona();

Estudiante.prototype.estudiar = function () {
      console.log("Estudiando");
}

let estudiante1 = new Estudiante("2328", "Juan", "X2134d");
estudiante1.saludar();
estudiante1.estudiar();
console.log(estudiante1.nombre);
```

Como podemos ver, la herencia se ha ejecutado correctamente pero nos queda aún un paso más. Ahora deberíamos hacer una prueba más para ver qué nos devuelve el constructor del objeto “estudiante1” recién creado:

```
console.dir( estudiante1.constructor );
```

Y lo que nos devuelve es un valor incorrecto de constructor: nos dice que el constructor de un estudiante es Persona():



Para terminar debemos hacer una corrección más para asignar correctamente el constructor Estudiante() del objeto Estudiante() mediante la siguiente línea:

```
Estudiante.prototype.constructor = Estudiante ;
```

Encapsulación

En programación orientada a objetos se denomina encapsulación a la capacidad de datos y funciones dentro de una clase. De esta manera sólo serán accesibles desde dentro de la clase, es más, no serán accesibles incluso desde instancias de esa misma clase.

En Javascript, **todas las variables y funciones declaradas dentro de una función constructor son solo accesibles desde esta misma función constructora**.

Si quisiéramos hacer accesibles a esas funciones o propiedades accesibles desde instancias de objetos construidos a través de esa misma función constructo **tendríamos que usar la palabra clave “this”.**

La palabra clave **“this”** se dice que es es capaz de convertir a variables y funciones de una función constructor en “propiedades” y “métodos” de ese objeto creado.

Veamos un ejemplo de una “función privada”:

```
 <script>
      function Caja(ancho, largo, alto) {
            //Función privada:
            function volumen(a,b,c) {
                  return a*b*c;
            }
      }
</script>
```

Esta función “volumen” es un método privado el cual no podrá ser usado excepto dentro del ámbito de la función constructora Caja. Bien, pues veremos ahora una posible forma de poder acceder a esta función de forma que podamos seguir aplicando el concepto de encapsulación:

```
<script>
      function Caja(ancho, largo, alto) {
            //Función privada:
            function volumen(a,b,c) {
                  return a*b*c;
            }
            this.calculoVolumen = volumen(ancho,largo,alto);
      }

      let contenedor =  new Caja(5,4,3);
      //volument(3,2,1)    --> No funcionaría
      //contenedor.volumen(3,2,1)    -->  No funcionaría
      console.log("Volumen de la caja: " + contenedor.calculoVolumen )
</script>
```

Método 2: Construyendo objetos mediante el Object Create.

Creando objetos y herencia con **Object.create()**

En **EcmaScript5** se introdujo un nuevo método llamado **Object.create()** el cual nos permite crear objetos y hacer instanciaciones de estos objetos haciendo también uso de este mismo método. Cuando usamos este nuevo método no tendremos que hacer uso del operador **“new”.** Esta nueva aproximación no es ni mejor ni peor que la que ya hemos estudiado, pero es interesante conocer esta forma de crear objetos debido a que existen multitud de recursos en internet que harán uso del mismo y quizás podamos nosotros empezar a usarla también. Con un ejemplo veremos más claro cómo podemos modificar el código del ejemplo anterior para usar esta nueva aproximación:

```
<script>
      //Objeto persona:
      let Persona = {
```

```
            //Propiedades
            dni : "" ,
            nombre : "" ,
            amigos : null ,
            //Función constructora: es obligatoria!!!
            init  : function(dni, nombre) {
                  this.dni = dni;
                  this.nombre = nombre;
                  this.amigos = new Array();
                  return this;  // Muy importante devolver el objeto mismo!!!
            } ,

            saludar : function () {  console.log ("Me llamo " + this.nombre); }
      };

      //Ahora vamos a crear el objeto Estudiante que hereda de Persona:
      let Estudiante = Object.create(Persona);
      Estudiante.numeroMatricula = "";  // Añadimos una nueva propiedad a Estudiante
      //Función constructora de Estudiante:
      Estudiante.init = function (dni, nombre, numeroMatricula) {
            Persona.init.call(this, dni, nombre); // Llamamos al método constructor
de Persona.
            this.numeroMatricula = numeroMatricula;
            return this; // Muy importante devolver el objeto mismo!!!
      }
      //Ahora vamos a añadir un nuevo método a Estudiante.
      Estudiante.estudiar = function () { console.log("Mi matricula es " +
this.numeroMatricula); }

      // Y AHORA VAMOS A INSTANCIAR UN OBJETO
      let estudiante2 = Object.create(Estudiante).init("123", "Pepe", "12");
      estudiante2.saludar();
      estudiante2.numeroMatricula = 222;
      estudiante2.estudiar();
      </script>
```

Método 3: Construyendo objetos mediante clases.

Clases en ES6

En la versión reciente de Javascript ES6 una de las novedades más importantes fue la incorporación de la declaración de **clases**. Mencionar que a día de hoy muchas de las características descritas en ES6 aún no han sido implementadas en todos los navegadores por lo que debemos de andarnos cautos a la hora de utilizar estas nuevas características.

Dicho esto, podemos mencionar que la construcción de objetos usando “clases” es mucho más limpia y simple y muy similar al modo en cómo se realizan en otros lenguajes como Java o C++.

Declaración e instanciación de clases:

En la siguiente línea de código podemos ver la declaración de una clase y su método constructor.

```
class Coordenada {
  constructor(x, y) {
      this.x = x;
      this.y = y;
  }
}
```

Nota: No puede haber más de un método llamado **constructor** en una clase de Javascript. Si por despiste colocamos más de uno recibiremos un mensaje de error "A class may only have one constructor".

Para **instanciar un objeto** de la anterior clase podemos utilizar el operador **new**:

```
var testCoordenada = new Coordenada(8,55);
```

Importante: **Declaraciones de clases siempre antes de su uso: ¡el orden sí importa!**

Es indispensable que se declaren las clases antes de usarlas. A diferencia de las funciones o variables en Javascript, que se pueden usar antes de haber sido declaradas, las clases deben conocerse antes de poder instanciar objetos. El siguiente código generaría un error:

```
var x = new MiClase();
class MiClase {};    // Primero necesitamos la definición
```

Expresiones de clases ES6

Otra forma de poder declarar clases en ES6 es mediante lo que se conoce como “expresiones”.
Básicamente consiste en crear una variable y asignarle una expresión definida mediante **class** y las llaves:

```
var Persona = class {
  constructor(nombre) {
      this.nombre = nombre;
  }
}
//Otro ejemplo:
```

```
var Coordena = class
{
   constructor(x,y) { this.x =x ; this.y = y; }
}
```

Creación de métodos en clases:

Las clases en ES6 pueden declarar sus métodos de una manera resumida, pues nos ahorramos la palabra **"function"** en la declaración de la función. Aunque también podemos usar la forma “tradicional” de creación de métodos.

```
class Coordenada {
  constructor(x, y) {
      this.x = x;
      this.y = y;
  }

  //Forma resumida
  esIgual(coordenada) {
      if (this.x == coordenada.x && this.y == coordenada.y) {
         return true;
      }
         return false;
  }

  //Forma tradicional
  imprimirCoordenadas = function () {
       console.log(`La x es ${this.x} y la y es ${this.y}`);
  }
}
```

Podríamos probar el código anterior con las siguiente líneas:

```
var testCoordenada = new Coordenada(8,55);
console.log(testCoordenada.esIgual(new Coordenada(3, 1)));
testCoordenada.imprimirCoordenadas();
```

Métodos estáticos en clases ES6

Esta nueva forma de crear objetos/clases también nos permite tener métodos estáticos. Haremos uso de la palabra reservada **“static”**. Como bien sabemos los métodos estáticos son aquellos que pueden ser utilizados sin necesidad de haber instanciado un objeto de la clase. Lo veremos mejor continuando el ejemplo anterior y creamos un método estático **definicion()**:

```
class Coordenada {
            constructor(x, y) {
                  this.x = x;
                  this.y = y;
            }
            //Forma resumida
            esIgual(coordenada) {
                  if (this.x == coordenada.x && this.y == coordenada.y) {
                  return true;
                  }
                  return false;
            };
            //Forma tradicional
            imprimirCoordenadas = function () {
                  console.log(`La x es ${this.x} y la y es ${this.y}`);
            }

            static definicion() {
                  console.log("Una coordenada se define por el valor de x e y");
            }
      }

      let coord1 = new Coordenada(12,32);
      console.log(coord1.esIgual(new Coordenada(12,32)));
      coord1.imprimirCoordenadas();
      Coordenada.definicion();
```

Métodos y Propiedades privadas en clases ES6

En EcmaScript 6 no hay soporte oficial para métodos ni propiedades privadas pero en la comunidad de desarrolladores existen diversas formas de "simular un comportamiento parecido" al de variables y funciones privadas.

Nosotros podemos optar por la siguiente solución:

- **Propiedades privadas:** declaramos las variables dentro de nuestro constructor sin utilizar la palabra reservada **"this"**. Para actualizar o acceder a dichas variables debemos de hacerlo a través de métodos públicos.
- **Métodos privados:** al igual que las propiedades, deben de ser declarados dentro del constructor sin usar "this".

¡¡¡ Para acceder a estas propiedades y métodos "privados" debemos crear funciones que los manipulen "dentro del mismo constructor" !!!!

Veamos un ejemplo:

```
 class User {

 constructor(name, password, email) {
        let name     = name;
        let password = password;
        let email    = email;

        let privado  = function() {
          console.log("Accediste a un método privado");
        }

        this.getPrivado = function(){ privado(); }
        this.getNombre = function(){return name;}
  }
}

var u = new User("usuario","123","usuario@example.com");
u.getPrivado();
console.log(u.name);
console.log(u.password);
console.log(u.getNombre());
```

La salida de este programa es:

```
Accediste a un metodo privado
undefined
undefined
usuario
```

Herencia con clases en ES6

Vamos a ver un ejemplo en que podremos observar el mecanismo de herencia haciendo uso de clases en ES6. Utilizaremos principalmente el **término "extends"** para heredar de otras clases, y haremos uso del **método super()** para llamar a los constructores de las clases padres. También con "**super" podremos acceder a los métodos de las clases padres.** Veamos el ejemplo:

```
<script>
      //Creamo una clase padre:
      class Criatura {
         constructor(nom) {
            this.nombre = nom;
            this.movimiento = 10;
            this.potencia = 1;
         }
```

```
        caminar() {
            console.log(this.nombre + ' camina ' + this.movimiento + ' metros');
        }

        atacar(objeto) {
            console.log(this.nombre + ' ataca a ' + objeto.nombre + ' y le causa ' +
this.potencia + ' puntos de vida');
        }

        diNombre() { console.log(“ Me llamo ” + this.nombre)}
    }


    //Creamos una clase Perro que hereda de Criatura
    class Perro extends Criatura {
        constructor()
        {
            super('Perro');
            this.potencia = 2;
        }

        saltar() {
            console.log(this.nombre + ' salta ' + this.movimiento / 5 + ' metros')
        }
    }

    //Creamos un segundo nivel de herencia ya que PerroZombi hereda de Perro
    class PerroZombi extends Perro {
        constructor() {
            super();
            this.nombre = 'Perro Zombi';
            super.saltar();
        }

        saltar() {
            console.log(this.nombre + ' intenta saltar... pero no puede')
        }
    }

    //Probamos nuestras clases:
    var perro = new Perro();
    var perroZombi = new PerroZombi();

    perro.caminar();
    perro.atacar(perroZombi);
    perro.saltar();
    perroZombi.atacar(perro);
    perroZombi.saltar();
```

```
      perroZombi.caminar();

console.log( 'PerroZombie es una instancia de \' Object? ' , perroZombi instanceof
Object ) ;
console.log( 'PerroZombie es una instancia de \' Cría? ' , perroZombi instanceof
Criatura ) ;
console.log( 'PerroZombie es una instancia de \' Perro? ' , perroZombi instanceof
Perro ) ;
console.log( 'PerroZombie es una instancia de \' PerroZombie? ' , perroZombi
instanceof PerroZombi );

</script>
```

Una cosa que debemos mencionar es que la función **super()** no solo nos sirve para acceder al constructor de la clase padre sino que también nos permite acceder a métodos de la clase padre desde la definición de las otras variables. Por ejemplo, podríamos llamar al método **diNombre**() de la siguiente manera:

```
class Perro extends Criatura {
            constructor()
            {
                super('Perro');
                this.potencia = 2;
                super.diNombre();   //Aquí vemos cómo se utiliza.
            }

            saltar() {
               console.log(this.nombre + ' salta ' + this.movimiento / 5 + '
metros')
            }
      }
```

Hay que remarcar que siempre que se quiera utilizar un constructor en una clase esta debe incluir el **método constructor( )**, que será llamado automáticamente en instanciar un nuevo objeto con el operador new.