JavaScript 03 – Objetos y Clases en Javascript

Introducción

En Javascript podemos definir cualquier variable como un objeto, existen dos formas para hacerlo:

- Declarándola con **new** (NO se recomienda)
- Forma recomendada: creando un *literal object* usando notación **JSON**.

Ejemplo con *new*:

```
let alumno = new Object();
alumno.nombre = 'Carlos';     // se crea la propiedad 'nombre' y se le asigna un valor
alumno['apellidos'] = 'Pérez Ortiz';    // se crea la propiedad 'apellidos'
alumno.edad = 19;
```

Creando un *literal object* según la forma recomendada, el ejemplo anterior sería:

```
let alumno = {
nombre: 'Carlos',
apellidos: 'Pérez Ortiz',
```

```
edad: 19,
};
```

Propiedades de un objeto

Podemos acceder a las propiedades con . (punto) o [ ]:

```
console.log(alumno.nombre);       // imprime 'Carlos'
console.log(alumno['nombre']);    // imprime 'Carlos'

let prop = 'nombre';
console.log(alumno[prop]);        // imprime 'Carlos'
```

Si intentamos acceder a propiedades que no existen no se produce un error, se devuelve *undefined*:

```
console.log(alumno.ciclo);      // muestra undefined
```

Sin embargo, se genera un error si intentamos acceder a propiedades de algo que no es un objeto:

```
console.log(alumno.ciclo);           // muestra undefined
console.log(alumno.ciclo.descrip);      // se genera un ERROR
```

En versiones anteriores de JavaScript, para evitar este error se comprobaba que existían las propiedades previamente. Veamos un ejemplo:

```
console.log(alumno.ciclo && alumno.ciclo.descrip);
// si alumno.ciclo es un objeto muestra el valor de
// alumno.ciclo.descrip y si no muestra undefined
```

Con ES2020 (ES11) se ha incluido el operador **?.** para evitar tener que comprobar esto nosotros:

```
console.log(alumno.ciclo?.descrip);
// si alumno.ciclo es un objeto muestra el valor de
// alumno.ciclo.descrip y si no muestra undefined
```

Este nuevo operador también puede aplicarse a **arrays**:

```
let alumnos = ['Juan', 'Ana'];
console.log(alumnos?.[0]);
// si alumnos es un array y existe el primer elemento muestra el valor
// si ese elemento no existe muestra undefined
// si no existe el objeto con el nombre alumnos da ERROR
```

Podremos recorrer las propiedades de un objecto con `for..in`:

```
for (let prop in alumno) {
console.log(prop + ': ' + alumno[prop])
}
```

Resultado:

```
for (let prop in alumno) {
    console.log(prop + ': ' + alumno[prop])
}
```

```
nombre: Carlos
apellidos: Pérez Ortiz
edad: 19
>
```

Una propiedad de un objeto puede ser una función:

```
alumno.getInfo = function() {
return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad +
'años'
}

console.log(alumno.getInfo()); //El alumno Carlos Pérez Ortiz tiene 19años
```

También se puede incluir la declaración del método en la declaración del objeto:

```
let alumno = {
nombre: 'Carlos',
apellidos: 'Pérez Ortiz',
edad: 19,
getInfo: function(){
     return 'El alumno ${this.nombre} ${this.apellidos} tiene ${this.edad} años';
    }
};

console.log(alumno.getInfo()); //El alumno Carlos Pérez Ortiz tiene 19 años
```

OJO: deberíamos poder ponerlo con sintaxis *arrow function*, pero no funciona.

```
let alumno = {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    edad: 19,
    getInfo: ()=> `El alumno ${this.nombre} ${this.apellidos} tiene ${this.edad} años`
};

console.log(alumno.getInfo()); //El alumno undefined undefined tiene undefined años
```

No funciona bien porque this tiene distinto valor dependiendo del contexto, y no se puede usar en estos casos con función flecha. Tienes un documento titulado "JavaScript - Anexo - Uso de this en contexto" que lo explica.

Si el valor de una propiedad es el valor de una variable que se llama como ella, desde ES2015 no es necesario ponerlo:

```
let nombre = 'Carlos'
let alumno = {
nombre,            // es equivalente a nombre: nombre
apellidos: 'Pérez Ortiz',
    ...
```

EJERCICIO: Crea un objeto llamado tvSamsung con las propiedades **nombre** (TV Samsung 42"), **categoria** (Televisores), **unidades** (4), **precio** (345.95) y con un método llamado **importe** que devuelve el valor total de las unidades (nº de unidades * precio).

Prueba el uso del método con un ejemplo.

Agregar y eliminar propiedades en un objeto:

Se pueden agregar propiedades sin más. Pero OJO!! Si nos equivocamos al modificar el valor de una propiedad (escribimos un nombre de propiedad distinto) nos va a crear una propiedad que no deseamos.

Para eliminar una propiedad se hace con la palabra reservada **delete**

```
let alumno = {
nombre: 'Carlos',
apellidos: 'Pérez Ortiz',
edad: 19,
};

//añadimos la propiedad email
alumno.email='cperez@email.com';
console.log(alumno.email); //cperez@email.com

//eliminamos la proiedad email
delete alumno.email;
console.log(alumno); // {nombre: 'Carlos', apellidos: 'Pérez Ortiz', edad: 19}
```

Métodos get y set en objetos

Al igual que en las clases (como veremos más adelante), se pueden declarar métodos get y set en una o en varias propiedades de un objeto para acceder a su contenido o para modificar su valor.

Clases en Javascript

Desde ES2015 la POO en Javascript es similar a como se hace en otros lenguajes: clases, herencia, cohesión, abstracción, polimorfismo, acoplamiento, encapsulamiento… En Javascript solo se permite un método constructor.

Más información en https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos

Veamos un ejemplo:

```
class Alumno {
    constructor(nombre, apellidos, edad) {
        this.nombre = nombre
        this.apellidos = apellidos
        this.edad = edad
    }
    getInfo() {
       return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad + ' años'
    }
}

let alumno1 = new Alumno('Carlos', 'Pérez Ortiz', 19)
console.log(alumno1.getInfo())   // imprime 'El alumno Carlos Pérez Ortíz tiene 19 años'
```

Recuerda que para las funciones y variables tipo var, da igual el orden (declaración primero y llamada después), o viceversa, pero para las clases no. Primero se declara la clase y luego se crean objetos.

Para las clases no se aplica el concepto de *hoisting*, por lo que no es posible crear objetos antes de declarar la clase.

EJERCICIO: Crea una clase Productos con las propiedades y métodos del ejercicio anterior (el de la TV). Además tendrá un método **getInfo** que devolverá: 'Nombre (categoría): unidades uds x precio € = importe €'. Crea 3 productos diferentes y prueba getInfo.

La palabra reservada this. Cuidado.

Dentro de una función se crea un nuevo contexto y la variable *this* pasa a hacer referencia a dicho contexto. Si en el ejemplo anterior hiciéramos algo como esto:

```
class Alumno {
    constructor(nombre, apellidos, edad) {
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
      }
    getInfo() {
        function nomAlum() {
          return this.nombre + ' '+ this.apellidos; // Aquí this no es el objeto Alumno
         }
         return 'El alumno' + nomAlum() + ' tiene '+ this.edad + ' años';
    }
}
```

Este código fallaría porque dentro de *nomAlum* la variable *this* ya no hace referencia al objeto Alumno sino al contexto de la función. Este ejemplo no tiene mucho sentido, pero a veces nos pasará en manejadores de eventos.

Si debemos llamar a una función dentro de un método (o de un manejador de eventos) tenemos varias formas de pasarle el valor de *this*:

1ª forma: Usando una *arrow function* que no crea un nuevo contexto, por lo que *this* conserva su valor.

```
getInfo() {
  let nomAlum=() => this.nombre + ' '+ this.apellidos;
  return 'El alumno ' + nomAlum() + ' tiene ' + this.edad + ' años';
}
```

2ª forma: Pasándole *this* como parámetro a la función:

```
getInfo() {
  function nomAlum(alumno){
     return alumno.nombre + ' ' + alumno.apellidos;
   }
   return 'El alumno ' + nomAlum(this) + ' tiene ' + this.edad + ' años';
}
```

3ª forma: Guardando el valor de *this* en otra variable (como *that*)

```
getInfo() {
   let that=this;
```

```
    function nomAlum() {
        return that.nombre + " " + that.apellidos;
    }
     return "El alumno " + nomAlum() + " tiene " + this.edad + " años";
}
```

4ª forma: Haciendo un *bind* de *this* , se explica en el apartado de eventos.

Herencia, sobreescritura, polimorfismo

Nota: Las relaciones entre las clases se dan por **Herencia** (Especialización o Generalización), **Asociación**, **Agregación** (por Referencia), **Composición** (por Valor) y **Dependencia** (Uso). Más información en Anexo - Introducción a UML en el apartado "2.5 Diagrama de Clases".

Una clase puede heredar de otra utilizando la palabra reservada **extends** y heredará todas sus propiedades y métodos.

Los métodos podemos sobrescribirlos en la clase hija (seguimos pudiendo llamar a los métodos de la clase padre utilizando la palabra reservada **super** (que es lo que haremos si creamos un constructor en la clase hija).

```
class Alumno {
  constructor(nombre, apellidos, edad) {
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.edad = edad;
  }
  getInfo() {
    return (
      "El alumno " + this.nombre + " " + this.apellidos + " tiene " +  this.edad + " años"
    );
  }
}

class AlumnInf extends Alumno {
  constructor(nombre, apellidos, edad, ciclo) {
    super(nombre, apellidos, edad);
    this.ciclo = ciclo;
  }
  getGradoMedio() {
    if (this.ciclo.toUpperCase() === "SMR") return true;
    else return false;
  }
  getInfo() {
    return (
      super.getInfo() + " y estudia el Grado " +
       (this.getGradoMedio() ? "Medio" : "Superior") + " de " + this.ciclo );
  }
}

let azp= new Alumno("Ana", "Zubiri Peláez", 24);
console.log(azp.getInfo()); // imprime: 'El alumno Ana Zubiri Peláez tiene 24 años'

let cpo = new AlumnInf("Carlos", "Pérez Ortiz", 19, "DAW");
```

```
console.log(cpo.getInfo()); // imprime 'El alumno Carlos Pérez Ortíz tiene 19 años y
estudia el Grado Superior de DAW'
```

En el ejemplo anterior hemos sobrescrito el método *getInfo*(), lo que implica que hay dos versiones de este método (la primera en la clase padre y la segunda en la clase hija). Se aplicará la primera versión de *getInfo*() o la segunda, dependiendo de si el objeto en cuestión es de una clase o de otra.

Siguiendo con el ejemplo visto anteriormente:

- Quitamos la salida por consola de ambos objetos
- Declaramos una función **imprimir()** fuera de las dos clases.
- Llamamos a la función imprimir pasando como argumento cada objeto

Al llamar a imprimir pasando como argumento cada objeto (cada uno es de un tipo), conseguimos que se llame, a su vez, al método **getInfo**() de la clase de ese objeto.

```
class Alumno {
  constructor(nombre, apellidos, edad) {
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.edad = edad;
  }
  getInfo() {
    return (
      "El alumno " + this.nombre + " " + this.apellidos + " tiene " +  this.edad + " años"
    );
  }
}

class AlumnInf extends Alumno {
  constructor(nombre, apellidos, edad, ciclo) {
    super(nombre, apellidos, edad);
    this.ciclo = ciclo;
  }
  getGradoMedio() {
    if (this.ciclo.toUpperCase() === "SMR") return true;
    else return false;
  }
  getInfo() {
    return (
      super.getInfo() + " y estudia el Grado " +
       (this.getGradoMedio() ? "Medio" : "Superior") + " de " + this.ciclo );
  }
}

let azp= new Alumno("Ana", "Zubiri Peláez", 24);
let cpo = new AlumnInf("Carlos", "Pérez Ortiz", 19, "DAW");

function imprimir(objeto){
    console.log(objeto.getInfo());
}
```

```
imprimir(azp);
imprimir(cpo);
```

Es lo que se llama **polimorfismo**, pues hay múltiples formas en tiempo de ejecución, y se ejecutará un método u otro dependiendo de la referencia pasada como argumento.

InstanceOf

Para saber el tipo con el que trabajamos se utiliza **instanceof**

Así sabremos el tipo de objeto que estamos recibiendo.

```
function imprimir(objeto){
    console.log(objeto.getInfo());
    if (objeto instanceof AlumnInf){
        console.log(objeto.nombre + ' es un alumno de informática');
    }
}
```

En el ejemplo anterior, preguntamos si el objeto pasado como argumento es un tipo de la clase **`AlumnInf`**, de ser así, muestra por consola la frase "Fulanito es un alumno de informática".

Hay que tener en cuenta que los objetos de la clase **`AlumnInf`** son también de la clase **`Alumno`** y también de la clase **`Object`**. Así que, para preguntar por el tipo de un parámetro, se suele hacer empezando por el tipo de menor jerarquía (el hijo).

EJERCICIO: crea una clase Televisores que hereda de Productos y que tiene una nueva propiedad llamada **tamaño**. El método **getInfo** mostrará el tamaño junto al nombre.

Métodos estáticos

Desde ES2015 podemos declarar métodos estáticos, pero no propiedades estáticas. Estos métodos se llaman directamente **utilizando el nombre de la clase** y no tienen acceso al objeto *this* (ya que no hay objeto instanciado).

```
class User {
static getRoles() {
return [”user”, “guest”, “admin”]
    }
}

console.log(User.getRoles()) // [”user”, “guest”, “admin”]
let usuario = new User(“john”)
console.log(usuario.getRoles()) // Uncaught TypeError: usuario.getRoles is not a function
```

El siguiente ejemplo demuestra varias cosas:

1. Cómo se implementa método estático en una clase,
2. Que una clase con un miembro estático puede ser sub-claseada.
3. Finalmente demuestra cómo un método estático puede (y cómo no) ser llamado.

```
class Tripple {
   static tripple(n) {
      n = n || 1;
      return n * 3;
```

```
    }
}

class BiggerTripple extends Tripple {
    static tripple(n) {
        return super.tripple(n) * super.tripple(n);
    }
}

console.log(Tripple.tripple());          // 3
console.log(Tripple.tripple(3));         // 9
console.log(BiggerTripple.tripple(3));   // 81
let tp = new Tripple();
console.log(tp.tripple()); //ERROR Logs 'tp.tripple is not a function'.
```

Atributos estáticos

Se pueden definir atributos estáticos con la palabra reservada **static**, y será un atributo que pertenece a la clase y no al objeto.

Para hacer referencia (utilizar este atributo) hay que poner *NombreDeClase.atributo*

Es muy útil para llevar un contador de objetos de la clase, o para llevar un atributo de clave única autoincremenetal (que en nuestro caso hemos llamado *id*).

```
class Alumno {
    static contadorObjetosAlumno = 0;
    constructor(nombre, apellidos, edad) {
        this.id = ++ Alumno.contadorObjetosAlumno; // cada vez que se crea un objeto,
                                                   // se incrementa
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
        console.log(this);
    }
}

let misAlumnos=[];
misAlumnos[0] = new Alumno('Carlos', 'Pérez Ortiz', 19);
misAlumnos[1] = new Alumno('Rosa', 'Cifuentes Martín', 24);
misAlumnos.push(new Alumno('Ana', 'Zubiri Peláez', 29));
```

```
▶ Alumno {id: 1, nombre: 'Carlos', apellidos: 'Pérez Ortiz', edad: 19}
▶ Alumno {id: 2, nombre: 'Rosa', apellidos: 'Cifuentes Martín', edad: 24}
▶ Alumno {id: 3, nombre: 'Ana', apellidos: 'Zubiri Peláez', edad: 29}
```

Este atributo contador también se hereda a las clases hijas de Alumno si las hubiere. Cada vez que se crea un objeto de una clase hija, incrementa este contador.

Resumiendo: Hemos creado una variable que tiene un identificador único para cada objeto que se crea, utilizando una variable estática.

Debido a que la variable estática se asocia con la clase y no con los objetos, permite asignar un valor único. Que en el ejemplo va preincrementándose para lograr que el id sea único y consecutivo, independientemente de que sea un objeto de la clase padre (donde está definido) o en la clase que hereda.

Constantes estáticas utilizando métodos estáticos

Para disponer de una variable estática de solo lectura, no podemos utilizar la palabra reservada **const.**

Lo que haremos es crear un método estático que solo permitirá leer el valor que devuelve el método, sin poder modificar este valor. Así parecerá que es una constante estática.

En el ejemplo siguiente declararemos la constante MAX_OBJ con el valor 5. (el máximo número de objetos que podremos crear serán 5, tanto de la clase padre como de la hija).

```
class Alumno {
    static contadorObjetosAlumno = 0;
    static get MAX_OBJ(){
        return 5;
    }
    constructor(nombre, apellidos, edad) {
        this.id = ++ Alumno.contadorObjetosAlumno; // cada vez que se crea un objeto,
                                                   // se incrementa
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
        console.log(this);
    }
}

console.log(Alumno.MAX_OBJ); // 5
Alumno.MAX_OBJ = 10;
console.log(Alumno.MAX_OBJ); // 5
```

Como hemos definido un método **get**, podemos recuperar esta información de la forma **Alumno.MAX_OBJ**

Como no hemos definido método **set** para este atributo, no podremos modificar el valor.

Aunque no hemos creado una variable, este método funciona como si fuera una variable (constante).

Funciona igual que un atributo de la clase.

Siguiendo con el ejemplo: cambiamos el constructor:

```
class Alumno {
    static contadorObjetosAlumno = 0;
    static get MAX_OBJ(){
        return 5;
    }
    constructor(nombre, apellidos, edad) {
        if(Alumno.contadorObjetosAlumno < Alumno.MAX_OBJ){
            this.id = ++ Alumno.contadorObjetosAlumno;
        } else {
```

```
            console.log('Se ha superado el máximo de objetos permitidos de la clase
Alumno');
        }
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
        console.log(this);
    }
}

let misAlumnos=[];
misAlumnos[0] = new Alumno('Carlos', 'Pérez Ortiz', 19);
misAlumnos[1] = new Alumno('Rosa', 'Cifuentes Martín', 24);
misAlumnos[2] = new Alumno('Ana', 'Zubiri Peláez', 29);
misAlumnos[3] = new Alumno('Rosa', 'Cifuentes Martín', 24);
misAlumnos[4] = new Alumno('Ana', 'Zubiri Peláez', 29);
misAlumnos[5] = new Alumno('Ana', 'Zubiri Peláez', 29);
```

Importante a tener en cuenta: En el caso anterior se crearán 6 alumnos, el sexto alumno se ha creado porque no hemos mandado llamar a una excepción, pero no tiene atributo id porque así lo hemos decidido en el constructor:

```
▶ Alumno {id: 1, nombre: 'Carlos', apellidos: 'Pérez Ortiz', edad: 19}
▶ Alumno {id: 2, nombre: 'Rosa', apellidos: 'Cifuentes Martín', edad: 24}
▶ Alumno {id: 3, nombre: 'Ana', apellidos: 'Zubiri Peláez', edad: 29}
▶ Alumno {id: 4, nombre: 'Rosa', apellidos: 'Cifuentes Martín', edad: 24}
▶ Alumno {id: 5, nombre: 'Ana', apellidos: 'Zubiri Peláez', edad: 29}
Se ha superado el máximo de objetos permitidos de la clase Alumno
▶ Alumno {nombre: 'Ana', apellidos: 'Zubiri Peláez', edad: 29}
misAlumnos
▶ (6) [Alumno, Alumno, Alumno, Alumno, Alumno, Alumno]
```

toString()

De forma implícita, todas las clases que vayamos a crear heredan (*extends*) de la clase *Object* . En Javascript, la clase *Object* es la clase padre de todas las clases, por tanto, podemos usar el método heredado *`toString`*() o sobrescribirlo.

Al convertir un objeto a string (por ejemplo, al concatenarlo con un String) se llama al método ***.toString()*** del propio objeto, que por defecto devuelve la cadena `[object Object]`. Podemos sobrecargar este método para que devuelva lo que queramos:

```
class Alumno {
  constructor(nombre, apellidos, edad) {
  this.nombre = nombre;
  this.apellidos = apellidos;
  this.edad = edad;
}

toString() {
```

```
   return this.apellidos + ', ' + this.nombre
 }
}

let cpo = new Alumno('Carlos', 'Pérez Ortiz', 19);
console.log('Alumno:' + cpo)     // imprime 'Alumno: Pérez Ortíz, Carlos'
                                 // en vez de 'Alumno: [object Object]'
```

Este método también es el que se usará si queremos ordenar una array de objetos (recuerda que *.sort()* ordena alfabéticamente para lo que llama al método *.toString()* del objeto a ordenar).

Por ejemplo, tenemos el array de alumnos *misAlumnos* que queremos ordenar alfabéticamente por apellidos. Si la clase *Alumno* no tiene un método *toString* habría que hacer como vimos en el tema de Arrays:

```
let misAlumnos=[];
misAlumnos[0] = new Alumno('Carlos', 'Pérez Ortiz', 19);
misAlumnos[1] = new Alumno('Rosa', 'Cifuentes Martín', 24);
misAlumnos[2] = new Alumno('Ana', 'Zubiri Peláez', 29);
misAlumnos.sort(function(alum1, alum2){
  if (alum1.apellidos > alum2.apellidos) return 1
  if (alum1.apellidos < alum2.apellidos) return -1
});
```

NOTA: como las cadenas a comparar pueden tener acentos u otros caracteres propios del idioma, el código anterior no siempre funcionará bien. La forma correcta de comparar cadenas es usando el método `.localeCompare()`. El código anterior debería ser:

```
misAlumnos.sort(function(alum1, alum2) {
  return alum1.apellidos.localeCompare(alum2.apellidos)
});
```

que con *arrow function* quedaría:

```
misAlumnos.sort((alum1, alum2) => alum1.apellidos.localeCompare(alum2.apellidos) )
```

o si queremos comparar por 2 campos ('apellidos' y 'nombre')

```
misAlumnos.sort((alum1, alum2) =>
(alum1.apellidos+alum1.nombre).localeCompare(alum2.apellidos+alum2.nombre) )
```

Si sobrescribimos el método toString, podemos utilizar este método para la ordenación:

```
class Alumno {
 constructor(nombre, apellidos, edad) {
   this.nombre = nombre;
   this.apellidos = apellidos;
   this.edad = edad;
 }

 toString() {
   return this.apellidos + ', ' + this.nombre
 }
}
```

```
let misAlumnos=[];
misAlumnos[0] = new Alumno('Carlos', 'Pérez Ortiz', 19);
misAlumnos[1] = new Alumno('Rosa', 'Cifuentes Martín', 24);
misAlumnos[2] = new Alumno('Ana', 'Zubiri Peláez', 29);
misAlumnos.sort(function(alum1, alum2){
  if (alum1.toString() > alum2.toString()) return 1;
  if (alum1.toString() < alum2.toString()) return -1;
});
```

Pero con el método *toString* que hemos definido antes podemos hacer directamente:

```
misAlumnos.sort()
```

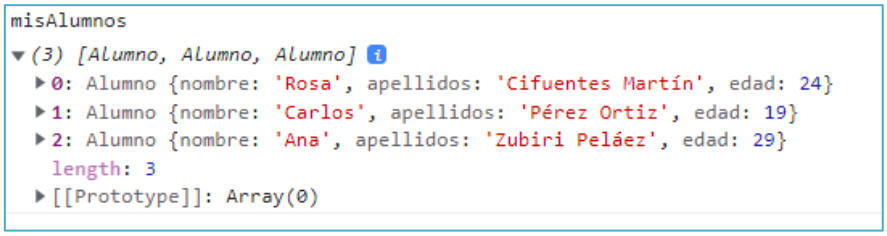
Quedando el ejemplo al completo:

```
class Alumno {
 constructor(nombre, apellidos, edad) {
   this.nombre = nombre;
   this.apellidos = apellidos;
   this.edad = edad;
 }

 toString() {
   return this.apellidos + ', ' + this.nombre
 }
}

let misAlumnos=[];
misAlumnos[0] = new Alumno('Carlos', 'Pérez Ortiz', 19);
misAlumnos[1] = new Alumno('Rosa', 'Cifuentes Martín', 24);
misAlumnos[2] = new Alumno('Ana', 'Zubiri Peláez', 29);
misAlumnos.sort();
```

Obteniendo en la consola:



NOTA: si queremos ordenar un array de objetos por un campo numérico lo más sencillo es restar dicho campo:

```
misAlumnos.sort((alum1, alum2) => alum1.edad - alum2.edad)
```

EJERCICIO: modifica las clases Productos y Televisores para que el método que muestra los datos del producto se llame de una manera más adecuada.

EJERCICIO: Crea 5 productos y guárdalos en un array. Crea las siguientes funciones (todas reciben ese array como parámetro):

prodsSortByName: devuelve un array con los productos ordenados alfabéticamente.

prodsSortByPrice: devuelve un array con los productos ordenados por precio.

prodsTotalPrice: devuelve el importe total de los productos del array, con 2 decimales.

prodsWithLowUnits: además del array recibe como segundo parámetro un número, y devuelve un array con todos los productos de los que quedan menos de las unidades indicadas.

prodsList: devuelve una cadena que dice 'Listado de productos:' y en cada línea un guión y la información de un producto del array.

valueOf()

Al comparar objetos (con >, <, …) se usa el valor devuelto por el método *.toString()*

Pero si definimos un método ***.valueOf()*** será este el que se usará en comparaciones:

```
class Alumno {
 ...
 valueOf() {
   return this.edad
 }
}

let cpo = new Alumno('Carlos', 'Pérez Ortiz', 19)
let aat = new Alumno('Ana', 'Abad Tudela', 23)
console.log(cpo < aat)     // imprime true ya que 19<23
```

Métodos get y set en JS

Los métodos get y set no son necesarios, pero sí recomendables para modificar y/o acceder al contenido de las propiedades.

Se pueden definir para una o varias propiedades de una clase. Incluso puede definirse únicamente uno de estos métodos (set o get).

Get: lee la información (valor) de la propiedad.
Set: pone información en la propiedad.
Estos métodos no pueden llamarse igual que la propiedad, por eso pondremos `_propiedad` (con guion bajo).

Con get y set no hay que poner paréntesis para llamar a los métodos:

Una vez definidos los métodos get y set para una propiedad, no usaremos paréntesis para invocar a esos métodos, accederemos como a sus propiedades:

- Para invocar al método get y que devuelva su valor haremos `= objeto.propiedad`

Para llamar al método set haremos: `objeto.propiedad = valor` Veamos un ejemplo para la propiedad nombre de la clase Alumno:

```
class Alumno {
 constructor(nombre, apellidos, edad) {
   this._nombre = nombre;
   this._apellidos = apellidos;
   this.edad = edad;
 }

get nombre(){
```

```
return this._nombre;
  }
set nombre(nombre){
this._nombre = nombre;
  }
}

let cpo =  new Alumno('Carlos', 'Pérez Ortiz', 19);
cpo.nombre = 'Carlos Luis'; //set nombre('Carlos Luis')
console.log (cpo.nombre); // get nombre() -> Carlos Luis
```

El método **set** es más necesario en el caso que queramos que se guarde la información con un formato determinado. Dentro del método **set** podemos hacer los cambios que necesitemos.

Por ejemplo, para que se guarde siempre en mayúsculas:

```
set nombre(nombre){
this._nombre = nombre.toUpperCase(); //
  }
```

El método **get** es muy útil para devolver información en un formato determinado como si accediéramos a una propiedad (sin paréntesis).

Por ejemplo, para devolver en formato: Apellidos, nombre utilizaremos el método get nombreCompleto()

```
get nombreCompleto(){
return this._apellidos + ', ' + this.nombre;
  }

console.log(cpo.nombreCompleto); // se accede sin paréntesis
```

Get y set, en conjunto, pueden ser una forma muy eficaz de obtener seguridad en cuando a validación de datos.

Agregar propiedades y métodos a los objetos de una clase (prototype).

En Javascript un objeto se crea a partir de otro (al que se llama *prototipo*). Así se crea una cadena de prototipos, el primero de los cuales es el objeto *null*.

Ya vimos como añadir una propiedad a un objeto. Pero si queremos que esa propiedad la tengan todos los objetos de la clase, y no queremos modificar el constructor, hay que hacerlo con **prototype**. Se le puede aplicar un valor por defecto.

Igualmente se pueden añadir métodos a la clase (también con **prototype**).

En el siguiente ejemplo se ha añadido la propiedad email y el método ApellidosMayusc() a la clase Alumnos.

```
class Alumno {
 constructor(nombre, apellidos, edad) {
   this.nombre = nombre;
   this.apellidos = apellidos;
   this.edad = edad;
 }
}

let cpo =  new Alumno('Carlos', 'Pérez Ortiz', 19);
```

```
Alumno.prototype.email='...@email.com'; // valor por defecto
console.log (cpo.email) // ...@email.com

Alumno.prototype.ApellidosMayusc= function () {
  return this.apellidos.toLowerCase();
}

console.log(cpo.ApellidosMayusc());
```

Prototipos y POO en JS5

Las versiones de Javascript anteriores a ES2015 no soportan clases ni herencia.

Este apartado está sólo para que comprendamos este código si lo vemos en algún programa, pero nosotros programaremos como hemos visto antes.

Si queremos emular en JS5 el comportamiento de las clases, para crear el constructor se crea una función con el nombre del objeto y para crear los métodos se aconseja hacerlo en el *prototipo* del objeto para que no se cree una copia del mismo por cada instancia que creemos:

```
function Alumno(nombre, apellidos, edad) {
  this.nombre = nombre
  this.apellidos = apellidos
  this.edad = edad
}
Alumno.prototype.getInfo = function() {
  return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad + ' años'
    }
}

let cpo = new Alumno('Carlos', 'Pérez Ortiz', 19)
console.log(cpo.getInfo())     // imprime 'El alumno Carlos Pérez Ortíz tiene 19 años'
```

Cada objeto tiene un prototipo del que hereda sus propiedades y métodos (es el equivalente a su clase, pero en realidad es un objeto que está instanciado).

Si añadimos una propiedad o método al prototipo se añade a todos los objetos creados a partir de él, lo que ahorra mucha memoria.

Uso de call y apply en Javascript

Con la palabra reservada **call** se puede llamar a un método de un objeto, desde otro objeto que no tiene ese método.

```
let cpo= {
  nombre: 'Carlos',
  apellidos: 'Pérez Ortiz',
  nombreCompleto: function(){
    return this.apellidos + ', '+ this.nombre;
  }
}
```

```
let azp= {
  nombre: ‘Ana’,
  apellidos: ‘Zubiri Peláez’,
}

console.log(cpo.nombreCompleto()); // Pérez Ortiz, Carlos
console.log(cpo.nombreCompleto.call(azp)); // Zubiri Peláez, Ana
```

Se pueden pasar argumentos a la llamada con **call**, en ese caso **la función llamada** tratará los argumentos como un parámetro normal.

```
console.log(cpo.nombreCompleto.call(azp, argum1, argum2));
```

Con **apply** se puede llamar a un método de un objeto con los datos de otro objeto que no tiene definido ese método. Se diferencia de call en cómo se pasan los argumentos.

En el caso de apply los argumentos se pasan en un array, no de uno en uno.

```
let cpo= {
  nombre: ‘Carlos’,
  apellidos: ‘Pérez Ortiz’,
  nombreCompleto: function(titulo, email){
     return  titulo + ‘ ‘ + this.nombre + ‘, ‘+ email;
  }
}

let azp= {
  nombre: ‘Ana’,
  apellidos: ‘Zubiri Peláez’,
}

arrayAgumentos= [’Doña’, ‘anzupe@email.com’];
console.log(cpo.nombreCompleto.apply(azp, arrayAgumentos));
```

La clase Object

Como todas las clases heredan de la clase Object, podemos utilizar con todos los objetos las propiedades y métodos de Object:

Ejemplo que lista los valores las propiedades de un objeto, también los métodos:

```
let cpo= {
  nombre: 'Carlos',
  apellidos: 'Pérez Ortiz',
  nombreCompleto: function(titulo, email){
    return  titulo + ' ' + this.nombre + ', '+ email;
  }
}

console.log(Object.values(cpo)); // (3) ['Carlos', 'Pérez Ortiz', ƒ]
```

Ejemplo que lista propiedades (clave: valor) de las propiedades y métodos de un objeto. En este caso, se utiliza *Object.prototype,* que permite añadir métodos a todas las clases:

```
Object.prototype.imprime = function() {
  Object.entries(this).forEach(([key, value])=>{
    console.log(key + ': '+value);
  })
}

let cpo= {
  nombre: 'Carlos',
  apellidos: 'Pérez Ortiz',
  nombreCompleto: function(titulo, email){
    return  titulo + ' ' + this.nombre + ', '+ email;
  }
}

cpo.imprime();
```

Podemos utilizar el método *hasOwnProperty*("nombre_propiedad") para que

```
console.log(cpo.hasOwnProperty("edad")); // devuelve false porque "edad" no es una propiedad
```

Este método está heredado de **Object**. Podemos ver otros métodos del prototipo Object expandiendo la salida del objeto desde la consola:

```
console.log(cpo);
```