JavaScript 03 – Objetos en Javascript

Introducción

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

- 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:

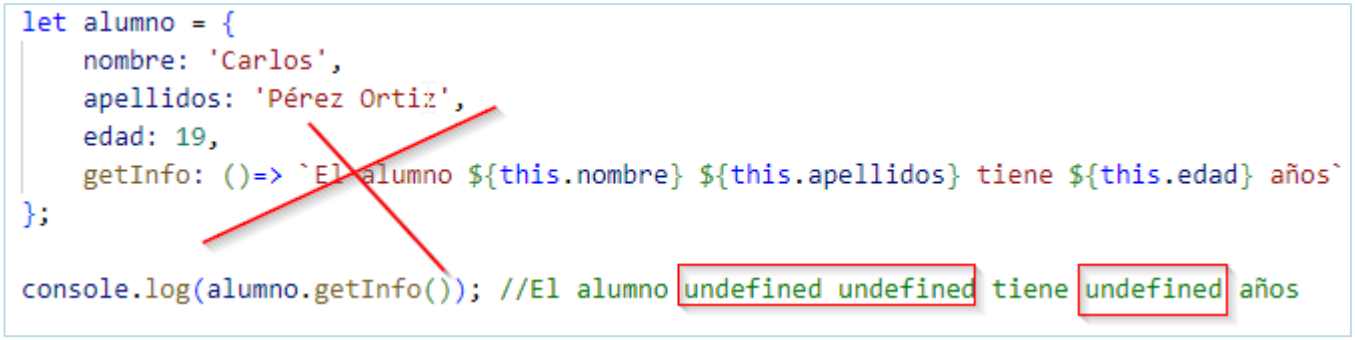
```
alumno.getInfo = function() {
    return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad +
'años'
```

```
}

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.



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.

Programación orientada a Objetos en Javascript

Desde ES2015 la POO en Javascript es similar a como se hace en otros lenguajes: clases, herencia, cohesión, abstracción, polimorfismo, acoplamiento, encapsulamiento…

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

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

Una clase puede heredar de otra utilizando la palabra reservada **extends** y heredará todas sus propiedades y métodos. Podemos sobrescribirlos en la clase hija (seguimos pudiendo llamar a los métodos de la clase padre utilizando la palabra reservada **super** -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 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'
```

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:

- Una de ellas es cómo un método estático es implementado en una clase,
- Otra es que una clase con un miembro estático puede ser sub-claseada.
- 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(6));         // 18
  console.log(BiggerTripple.tripple(3));   // 81
  var tp = new Tripple();
  console.log(tp.tripple()); //ERROR Logs 'tp.tripple is not a function'.
```

toString()

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;
});
```

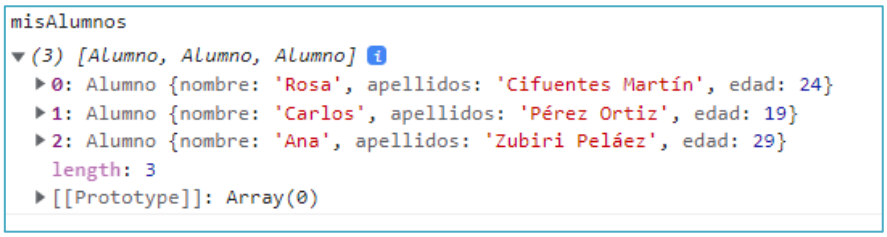
Pero con el método *toString* que hemos definido antes podemos hacer directamente:

```
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
```

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.

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*.

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.