NodeJS: Objetos

Gabriel Rodríguez Flores

October 5, 2021

ADA ITS
INSTITUTO TECNOLÓGICO SUPERIOR

    - Objectos y JSON (conversion)
- Destructuring
- Switch trick

Contents

1 Teoría

1.1 Objetos JSON

Referencia

Exactamente igual que en java, pero no necesitamos pre-definir los parámetros que va a tener de antemano.

Aunque siempre podremos también crear una clase, con sus constructor y métodos para replicar y crear automáticamente varios objetos de la misma estructura.

1.1.1 Manipulación

1.1.1.1 Inicialización Nuevo o vacío

```
var o1 = new Object(); // No se usa mucho
var o2 = {}; // Este es más recomendable
```

Con contenido directamente

```
var coche = {
    marca: 'seat',
    modelo: 'león',
    color: 'rojo'
};
```

1.1.1.2 Lectura/Escritura Para acceder a un parámetro podremos aceder de dos maneras:

```
objeto.propiedad
objecto['propiedad']
```

Para cada propiedad podremos usarlo como una variable normal, tanto para leer el valor como para escribirlo

```
console.log(coche.marca); // seat
coche.color = 'azul';
coche.asientos = 5; // Podemos asignar nuevas propiedades
console.log(coche.potencia); // undefined -- propiedades no existentes
```

tip

Podemos acceder a una propiedad de forma dinámica:

```
const key = 'marca';
console.log(coche[key]); // seat
```

También podemos crear propiedades con espacios:

```
coche['Caballos de potencia'] = 180;
```

1.1.2 Metodos

1.1.2.1 Métodos propios También podemos crear (como con Java) los métodos de un objeto, creando una función y asignándosela a la propiedad

```
const persona = {
    nombre: 'Gabri',
    apellido: 'Rodríguez',
    edad: 27,
    nombreCompleto: function () {
        return `${this.nombre} ${this.apellido}` // 'this' refiere al
            propio objeto
    }
};
```

Para invocarlo lo hacemos como una función normal accediendo a su propiedad

```
persona.nombreCompleto(); // Gabri Rodríguez
```

Si no indicamos los paréntesis '()', en lugar de invocar la función estaremos recogiendo su definición, la función en sí y no el resultado.

```
persona.nombreCompleto; // function () { return `${this.nombre} ${this.
    apellido}` }
```

También podemos crear las funciones con la sintaxis de la función flecha, pero perderíamos el vínculo con el objeto, es decir, **no podríamos usar el this**.

```
const operaciones = {
    suma: (a,b) => a+b,
    resta: (a,b) => a-b,
    mult: (a,b) => a*b,
    div: (a,b) => a/b,
};
```

1.1.2.2 Métodos de la clase Object Existen más métodos pero estos son los más interesantes:

- assign()
- create()
- keys()
- values()

- entries()
- fromEntries()
- getOwnPropertyNames()
- is()

1.1.3 Desestructuración

Desestructuración Una funcionalidad muy usada de javascript es la descomposición de objectos, de manera que podemos extrar en distintas variables los objetos de manera rápida, muy parecido al operador '…' que vimos en Arrays y que también funciona con los objetos.

```
1 const persona = {
2     nombre: 'Gabri',
3     apellido: 'Rodríguez',
4     edad: 27,
5 };
6
7 const { nombre, edad } = persona; // Podemos recuperar solo las
     propiedades que necesitemos, no hace falta todas
8 console.log(`${nombre} tiene ${edad} años`);
```

- **Alias** Es posible que queramos cambiar el nombre de la propiedad que queremos recoger en la desestructuración, por lo tanto podemos recogerlo asignandole otro nombre de variable, usando: `{ key: alias }`

```
1 const { nombre: name, edad: years } = persona; // Podemos recuperar
     solo las propiedades que necesitemos, no hace falta todas
2 console.log(`${name} tiene ${years} años`);
```

- Otra característica útil, es aprovechar el mismo nombre de propiedad que de variable, y así reducir la escritura.

```
1 const nombre = 'Gabriel';
2 const años = 27;
3
4 const persona = {
5     nombre, // Como la key y la variable tienen el mismo nombre,
        podemos usar esta sintaxis, que es equivalente a: nombre: nombre
6     edad: años // Aquí usamos en contenido de la variable 'años' pero
        no es el mismo nombre que la propiedad.
7 };
```

- **Rest** Este operador puede ser utilizado para recoger el resto de parámetros que no se han definido en la desestructuración.

```
const persona = {
    nombre: 'Gabri',
    apellido: 'Rodríguez',
    edad: 27,
};

const { nombre, ...resto } = persona; // Podemos recuperar solo las
   propiedades que necesitemos, no hace falta todas
console.log(resto); // { apellido: 'Rodríguez', edad: 27 }
```

1.1.4 Tip y consejos

1.1.4.1 Switch trick

- Es posible, y se recomienda en la mayoría de los casos, realizar la lógica de selección de un *switch* mediante el uso de objetos. De la siguiente manera:

```
const option = 2;

const value = {
  1: 'value1',
  2: 'value2',
  3: 'value3',
}[option];

console.log(value); // value2
```

- También permite la recepción de opciones como *strings* y la selección de otros tipos de datos e incluso funciones

```
const opcion = 'mult';

const resultado = {
    suma: (a,b) => a+b,
    resta: (a,b) => a-b,
    mult: (a,b) => a*b,
    div: (a,b) => a/b,
}[opcion](3,7);

console.log(resultado); // 21
```

1.2 Clases

Se usa cuando queremos organizar o hacer un modelo de datos predefinido y reutilizarlo, automiatizandolo con constructor y métodos

1.2.1 Estructura

```
class MyClass {
  // class methods
  constructor() { ... }
  method1() { ... }
  method2() { ... }
  method3() { ... }
  ...
}

const o = new MyClass();
o.method1();
```

1.2.1.1 Constructor

```
class Rectangulo {
  constructor(alto, ancho) {
    this.alto = alto;
    this.ancho = ancho;
  }
}
```

> Nota: No hay sobrecarga de constructores ni de métodos como en Java, ya que aquí no hay tipado de datos. En Typescript hay algo parecido, en el que defines con cuantas y cuales variables puedes invocar a la función, pero siempre será la misma funcion. Por lo tanto sólo puedes definir un constructor completo e ir restando variables en orden.

1.2.1.2 Métodos Get/Set

```
class Rectangulo {
  constructor(alto, ancho) {
    this.alto = alto;
    this.ancho = ancho;
  }
  // Getter
  get area(){
    return this.calcArea();
  }
  // Setter
  set alto(alto){
    this.alto = alto;
  }
  // Method
  calcArea () {
    return this.alto * this.ancho;
  }
}
```

static

```
class Persona {
    constructor(nombre){
        this.nombre = nombre;
    }

    static saluda(){
        console.log('Hola !');
    }
}
/* No es necesario 'new' (crear objeto) */
Persona.saluda(); // Hola !
```

1.2.1.3 Extends

```
class Animal {
  constructor(nombre) {
    this.nombre = nombre;
  }

  hablar() {
    console.log(this.nombre + ' hace un ruido.');
  }
}

class Perro extends Animal {
  hablar() {
    console.log(this.nombre + ' ladra.');
  }
}
```

super

```
class Gato extends Animal {
    constructor(nombre){
        super(nombre);
    }

    hablar(){
        super.hablar();
    }
}
```

1.2.1.4 Singleton Una clase 'Singleton' es una clase que sólo puede ser instanciada una vez (sólo se puede crear un objeto).

Esto es de gran utilidad cuando quieres utilizar siempre el mismo objeto durante toda la aplicación. Como por ejemplo, realizar una conexión a una BBDD (que se realizaría en el constructor), y luego sólo

usar sus métodos para interactuar. Así evitamos abrir varias conexiónes.

La forma de crear una clase Singleton es javascript es comprobando en el constructor si la clase ya ha sido instanciada:

En NodeJS basta con exportar la instancia (objeto) y no la clase en sí, y NodeJS mantiene el objeto cacheado y cada vez que se requiera esa clase, devolverá el mismo objeto.

```
class SingletonClass {
    constructor(name) {
        this.name
    }
}
module.exports = new SingletonClass();
// en lugar de: module.exports = SingletonClass
```

2 Ejemplos

- Eliminación de una propiedad

```
/* Dado ese objeto */
const persona = {
    nombre: 'Gabri',
    apellido: 'Rodríguez',
    sexo: 'H',
    edad: 27,
    altura: 1.75,
    peso: 75
};

/* Opción 1 */
delete persona.peso;
/* Opción 2 */
const { peso, ...nuevaPersona } = persona;
```

- Añadir unir uno o varios objetos

```
/* Dado ese objeto */
const persona = {
    nombre: 'Gabri',
    apellido: 'Rodríguez',
};
const datos = {
    sexo: 'H',
    edad: 27,
    altura: 1.75,
    peso: 75
```

```
};

/* Opción 1 */
const nuevaPersona = { ...persona, ...datos};
/* Opción 2 */
Object.assing(persona, datos); // Se guarda todo en persona
```

3 Ejercicios

- Básicos: Ejercicios de W3School sobre objetos

3.1 Básico

- Función que reciba un objeto `{ bien: N, mal, M }` y devuelva el resultado de la resta
  - enlace
- Función que reciba las dimensiones de un cubo en un objeto y devuelva el volumen `{ ancho: 2, largo: 5, alto: 1 }`
  - enlace
- Recoger el resto de un objeto en la variable 'rest'. Sólo hay que editar el código
  - enlace

3.2 Fácil

- Transformar un objeto en un array, cada elemento será clave-valor: `{ a: 1, b: 2 })--> [["a", 1], ["b", 2]]`
  - enlace
- Transformar un objeto en un array con dos elementos [claves, valores] : `{ a: 1, b: 2, c: 3 })--> [["a", "b", "c"], [1, 2, 3]]`
  - enlace

3.3 Medio

- Función que invierta clave y valor `{ "z": "q", "w": "f"})--> { "q": "z", "f": "w" }`

  - enlace

- Función que recoge un array de objetos con el nombre del alumno y un array de notas `{ name: 'John', notes: [3, 5, 4]}`, y devuelve el nombre y la nota más alta `{ name: "John", topNote: 5 }`

  - enlace

3.4 Dificil

- Comprobar si dos objetos son iguales de manera recursiva.

  - enlace

4 Entregables

4.1 En clase

- Realizar los ejercicios básicos

4.2 Tarea

1. Realizar todos los ejercicios Fácil y Medio
2. Realizar el ejercicio Dificil