Tema 3.1.
Objetos de Datos

- Declaración de Objetos
- JSON
- Clases
- Desestructuración de Objetos
- Iteración de Objetos

Declaración de Objetos

En Javascript, existe un tipo de dato llamado **objeto** .

Declaración de un objeto

```
const objeto = new Object(); // Evitar esta sintaxis en Javascript (no se suele usar)
```

En Javascript, siempre que podamos, se prefiere utilizar la **notación literal (**las llaves {}), una forma abreviada para crear objetos, sin necesidad de utilizar la palabra new.

```
const objeto = {}; // Esto es un objeto vacío
const player = {
        name: "Mario",
        life: 100,
        power: 10,
};
```

Estas variables dentro de los objetos se suelen denominar **propiedades**.

Declaración de Objetos

Propiedades de un objeto: acceder a sus propiedades de dos formas:

```
// Notación con puntos (preferida)
console.log(player.name); // Muestra "Mario"
console.log(player.life); // Muestra 100

// Notación con corchetes
console.log(player["name"]); // Muestra "Mario"
console.log(player["life"]); // Muestra 100
```

Suele ser la **notación con puntos**, mientras que la notación con corchetes se suele conocer en otros lenguajes como «arrays asociativos» o «diccionarios».

OJO: Hay ciertos casos en los que sólo se puede utilizar la **notación con corchetes**, como por ejemplo cuando se utilizan espacios en el nombre de la propiedad.

Declaración de Objetos

Añadir propiedades

Podemos añadir **propiedades** al **objeto** después de haberlo creado:

```
// FORMA 1: A través de notación con puntos
const player = {};
player.name = "Mario";
player.life = 100;
player.power = 10;
/
/ FORMA 2: A través de notación con corchetes
const player = {};
player["name"] = "Mario";
player["life"] = 99;
player["power"] = 10;
```

Las **propiedades** del objeto pueden ser utilizadas como variables. De hecho, se usan los objetos como elementos para organizar múltiples variables.

Declaración de Objetos

Métodos de un objeto

Si dentro de una variable del objeto metemos una función (*o una variable que contiene una función*), tendríamos lo que se denomina un **método de un objeto**:

```
const user = {
        name: "Mario",
        talk: function() { return "Hola"; }
};
user.name; // Es una variable (propiedad), devuelve "Mario"
user.talk(); // Es una función (método), se ejecuta y devuelve "Hola"
```

Esto es muy similar a un concepto que veremos más adelante llamado Clase

Declaración de Objetos

Cuando se necesita acceder desde un método de un objeto a las propiedades del mismo se puede usar el objeto **this**, en lugar del propio objeto

Ejemplo anterior:

```
    mostrar:function(){
        console.log(`${this.origen} a ${this.destino}`);
        console.log(this.precio);
    }
};
```

Declaración de Objetos

¿Qué aporta this? Seguridad.

Ejemplo: si se hubiera asignado viaje a otro objeto, ejemplo oferta, daría un error al acceder a las propiedades del objeto this;

```
let viaje={
    origen:"Granada",
    destino:"Londres",
    dias:80,
    precio:500,
    mostrar:function(){
        console.log(`${viaje.origen} a ${viaje.destino}`);
        console.log(viaje.precio);
    }
};
let oferta=viaje;
viaje=null;
oferta.mostrar();
```

```
Uncaught TypeError: viaje is null
    mostrar      http://localhost/cliente/letConst.js:7
    <anonymous>  http://localhost/cliente/letConst.js:13
[Saber más]
```

Declaración de Objetos

Si en lugar de esos, se usase this:

```
let viaje={
    origen:"Granada",
    destino:"Londres",
    dias:80,
    precio:500,
    mostrar:function(){
        console.log(`${this.origen} a ${this.destino}`);
        console.log(this.precio);
    }
};
let oferta=viaje;
viaje=null;
oferta.mostrar();
```

```
Granada a Londres
500
```

JSON

JSON son las siglas de **JavaScript Object Notation**, y es un **formato ligero** de datos, con una estructura (*notación*) específica, que es totalmente compatible de forma nativa con Javascript. Se basa en la sintaxis que tiene Javascript para crear objetos.
Un archivo **JSON** mínimo suele tener la siguiente sintaxis:
{ }
Su contenido puede ser simplemente un Array, un Number, un String, un Booleano o incluso un NULL , sin embargo, lo más habitual es que parta siendo un Object o un Array. Se puede comprobar en https://jsonlint.com/ si algo concreto es un **JSON** válido o no.
Un archivo JSON suele contener mucha información almacenada. Vamos a modificar ese objeto vacío para que contenga más datos para ejemplificarlo:

```
{ "name": "Mario",
"life": 3,
"totalLife": 6
"power": 10,
"dead": false,
"props": ["invisibility", "coding", "happymood"],
"senses": { "vision": 50, "audition": 75, "taste": 40, "touch": 80 }
}
```

JSON

Si comparamos un **JSON** con un objeto Javascript, aparecen algunas ligeras diferencias y matices:

- Las propiedades del objeto deben estar entrecomilladas con «comillas dobles»
- Los textos deben estar entrecomillados con «comillas dobles»
- Sólo se puede almacenar tipos como String, Number, Object, Array,Boolean o null.
- Tipos de datos como Function, Date, Regexp u otros, no es posible almacenarlos en un JSON.
- Tampoco es posible añadir **comentarios** en un JSON.

Existe una extensión para Visual Code llamada Fix JSON que corrige los errores de formato de un JSON.

En Javascript tenemos una serie de métodos que nos facilitan la tarea de pasar de Object de Javascript a **JSON** y viceversa.

Convertir JSON a objeto: JSON.parse()

Parsear: convertir JSON a objeto Javascript. Es una acción que analiza un String que contiene un **JSON** válido y devuelve un objeto Javascript con dicha información correctamente estructurada.

```
const json = '{ "name": "Mario", "life": 99 }';
const user = JSON.parse(json);
user.name; // "Mario"
user.life; // 99
```

JSON

Convertir objeto a JSON: JSON.stringify()

Operación inversa: **convertir un objeto Javascript a JSON.** Se utiliza para transformar un objeto de Javascript a **JSON** rápidamente:

```
const user = {
        name: "Mario",
        life: 99,
        talk: function () { return "Hola!"; },
};
JSON.stringify(user); // '{"name":"Mario","life":99}'
```

Como las funciones no están soportadas por **JSON**, en el caso anterior devolverá un String omitiendo las propiedades que contengan funciones (*u otros tipos de datos no soportados*).

Se le puede pasar un **segundo parámetro** al método .stringify(), que será un array que actuará de filtro a la hora de generar el objeto. Ejemplo:

```
const user = { name: "Mario", life: 99, power: 10, };
JSON.stringify(user, ["life"]) // '{"life":99}'
JSON.stringify(user, ["name", "power"]) // '{"name":"Mario","power":10}'
JSON.stringify(user, []) // '{}' Ninguna propiedad
JSON.stringify(user, null) // '{"name":"Mario","life":99,"power":10}' Todas las propiedades
```

JSON

Leyendo JSON externo

Normalmente los contenidos **JSON** suelen estar almacenados en un archivo externo, que habría que leer desde nuestro código Javascript.

Para ello, hoy en día se suele utilizar la función fetch() para hacer peticiones a sitios que devuelven contenido JSON. También se podría leer ficheros locales con contenido .json.

Esto se verá más adelante cuando veamos las peticiones HTTP.

Clases

Una **clase** sólo es una forma de organizar código de forma entendible con el objetivo de simplificar el funcionamiento de nuestro programa. Las clases son «conceptos abstractos» de los que se pueden crear objetos de programación, cada uno con sus características concretas.
La clase es el **concepto abstracto** de un objeto, mientras que el **objeto** es el elemento final que se basa en la clase.

Clases

Constructores de clases

Es un método especial que permite crear e inicializar de forma personalizada un objeto a partir de una clase .

Sólo puede haber un constructor por clase. Se ejecuta inicializando las propiedades del objeto cuando se utiliza el operador new.

```
class viaje {
        origen="Granada";
        destino="Londres";
        dias=80;
        precio=500;
        constructor(or,des,di,pre){
            this.origen=or;
            this.destino=des;
            this.dias=di;
            this.precio=pre;
        }
        mostrar(){
            console.log(`${this.origen} a ${this.destino}`);
            console.log(this.precio);
        }
    };
    let miViaje=new viaje("Barcelona","Ibiza",2,112);
    miViaje.mostrar();
```

```
Barcelona a Ibiza
112
```

Clases

Propiedades Públicas y Privadas

Por defecto, las propiedades establecidas en una clase son públicas, es decir, son accesibles desde fuera de la clase:

```
class viaje {
        origen="Granada";
        destino="Londres";
        constructor(or,des){
            this.origen=or;
            this.destino=des; }};
let miViaje=new viaje("Barcelona","Ibiza");
console.log(miViaje.origen);//Muestra Barcelona
```

Desde ECMAScript 2020, se pueden crear propiedades privadas, visibles sólo dentro de la clase, anteponiéndoles el carácter #:

```
class viaje {
        #origen="Granada";
        destino="Londres";
        constructor(or,des){
            this.#origen=or;
            this.destino=des; }};
let miViaje=new viaje("Barcelona","Ibiza");
console.log(miViaje.origen);//Muestra Undefined
console.log(miViaje.#origen);//Da error
miViaje.mostrar();Muestra los datos correctamente
miViaje.origen="Albacete"; //Funcionaría, creando una propiedad nueva
```

Clases

Ámbitos de propiedades de clase

Ámbito dentro de un método. Si declaramos propiedades dentro de un método con let o const, estos elementos existirán sólo en el método en cuestión. No serán accesibles desde fuera del método:

```
class Personaje {
        constructor() {
                const name = "Mario";
                console.log("Constructor: " + name); }
        method() { console.log("Método: " + name); }
}
const c = new Personaje(); // 'Constructor: Mario'
c.name; // undefined
c.method(); // 'Método: '
```

La variable name solo se muestra cuando se hace referencia a ella dentro del constructor() que es donde se creó y el ámbito donde existe.

Clases

Ámbito de clase. Podemos crear propiedades dentro de la clase o dentro del constructor con this: en ambos casos las propiedades tendrán alcance en toda la clase:

```
class Personaje {
        name = "Mario"; // ES2020+
        constructor() {
                this.name = "Mario"; // ES2015+
                console.log("Constructor: " + this.name);
        }
        metodo() {
                console.log("Método: " + this.name);
         }
}
const c = new Personaje(); // 'Constructor: Mario'
c.name; // 'Mario'
c.metodo(); // 'Método: Mario'
```

Si quieres que estas propiedades de clase se puedan modificar desde fuera de la clase, añade el # antes del nombre de la propiedad al declararla. De esta forma serán propiedades privadas, y sólo se podrá acceder a ellas desde el interior de los métodos de la clase.

Clases

Métodos Públicos y Privados

Funcionan igual que las propiedades públicas y privadas, y se definen igual.
Ejemplo de métodos privados:

```
class Personaje {
        name = "Mario";
        constructor() {
                this.#hablar();
        }
        #hablar() {
                console.log(“Soy Mario!");
        }
}
const mario = new Personaje(); //Soy Mario! (se ha accedido a #hablar() desde dentro de la clase)
mario.#hablar(); // Da error, no se puede acceder a un método privado desde fuera de la clase //
Uncaught SyntaxError
mario.hablar; //Error is not a function mario.hablar();
```

Clases

¿Qué es un método estático?

Para utilizar el método como método(), debemos crear previamente el objeto basado en la clase haciendo un new Personaje().

Nos podría interesar crear **métodos estáticos** en una clase, ya que este tipo de métodos **no requieren crear una instancia**, sino que se pueden ejecutar directamente sobre la clase:

```
class Personaje {
        name = "Mario"; // ES2020+
        constructor() {
                this.name = "Mario"; // ES2015+
                console.log("Constructor: " + this.name);
        }
        static metodo() {
                console.log(“Hola”);
         }
}
Personaje.metodo(); // Método estático (no requiere instancia) “Hola”
const c= new Personaje(); // Creamos una instancia
c.metodo(); // Uncaught TypeError
```

Una de las limitaciones de los **métodos estáticos** es que en su interior sólo podremos hacer referencia a elementos que también sean estáticos.

Clases

Herencia de clases

Si una clase hereda de otra, se puede ejecutar el constructor de la clase madre usando **super()**

```
class Miembro {
        nombre="nombre apellido1 apellido2";
        edad=30;
        estado="activo";
        constructor(n,ed,es){
            this.nombre=n;
            this.edad=ed;
            this.estado=es;
        }
        cobrar(){
            console.log(`El miembro ${this.nombre} ha cobrado`);
        }
    };
    class Profesor extends Miembro{
        nAlumnos=0;
        constructor(n,ed,es,Nalu){
            super(n,ed,es);
            this.nAlumnos=Nalu;
        }
        cobrar(){
            console.log(`El profesor ${this.nombre} ha cobrado`);
        }
    }
```

Clases

```
let unMiembro=new Miembro("Juan",20,"activo");
unMiembro.cobrar();
let unProfesor=new Profesor("Pepe",60,"baja",25);
unProfesor.cobrar();
```

```
El miembro Juan ha cobrado
El profesor Pepe ha cobrado
```

Clases

También se puede utilizar super para llamar desde la clase hoja a un método de la clase padre:

```
class Padre {
        soloPadre() {
                console.log("Tarea en el padre...");
        }
        padreHijo() {
                console.log("Tarea en el padre...");
        }
        sobreHijo() {
                console.log("Tarea en el padre...");
        }
}
class Hijo extends Padre {
        padreHijo() {
                super.padreHijo();
                console.log("Tarea en el hijo...");
        }
        soloHijo() {
                console.log("Tarea en el hijo...");
        }
        sobreHijo() {
                console.log("Tarea en el hijo...");
         }
}
```

Clases

Clases en ficheros externos

Generalmente las clases se suelen almacenar en ficheros individuales, de forma que cada clase que creamos, debería estar en un fichero con su mismo nombre:

```
// Animal.js
export class Animal {
        /* Contenido de la clase */
        }
```

Luego, si queremos crear objetos basados en esta clase, lo habitual es importar el fichero de la clase en cuestión y crear el objeto a partir de la clase:

```
// index.js
import { Animal } from "./Animal.js";
const pato = new Animal();
```

Si nuestra aplicación se complica mucho, podríamos crear carpetas para organizar mejor aún nuestros ficheros de clases, y por ejemplo, tener la clase Animal.js dentro de una carpeta classes (*o similar*).

Desestructuración de Objetos

Utilizando la desestructuración de objetos podemos **separar** en variables las propiedades que teníamos en el objeto:

```
const user = { name: "Mario", role: "streamer", life: 99 }
const { name, role, life } = user;
console.log(name);
console.log(role, life);
```

También podríamos hacer:

```
console.log({ name, role, life });
```

Se puede volver a estructurar un objeto. Además, se puede renombrar las propiedades:

```
const { name, role: type, life } = user;
console.log({ name, type, life });
```

Cuando una de esas propiedades no exista (*o tenga un valor undefined*), también podemos establecerle un valor por defecto:

```
const { name, role = "normal user", life = 100 } = user;
console.log({ name, role, life });
```

Esto hará que, si no existe la propiedad role en el objeto user, se cree la variable role con el "normal user".

Desestructuración de Objetos

Reestructurando nuevos objetos

La **desestructuración** se puede usar para reutilizar objetos y **recrear nuevos objetos** a partir de otros, basándonos en objetos ya existentes, añadiéndole nuevas propiedades o incluso sobreescribiendo antiguas.

Ejemplo:

```
const user = { name: "Mario", role: "streamer", life: 99 }
const fullUser = { ...user, power: 25, life: 50 }
```

Creamos un nuevo objeto fullUser con las mismas propiedades de user, la nueva propiedad power y sobreescribimos la propiedad life con el valor 50.

Haciendo copias de objetos

Con valores primitivos (*números, strings, booleanos...*), en Javascript se pasan **por valor**.
Los valores más complejos (*no primitivos: objetos, arrays, etc...*) se **pasan por referencia**.

```
const user = {
        name: "Mario",
        role: "streamer",
        life: 99,
        features: ["learn", "code", "paint"]
}
const fullUser = { ...user, power: 25, life: 50 }
```

Cuando hacemos la desestructuración ...user, estamos separando todas las propiedades de user y añadiéndolas a nuestro fullUser una por una.

Desestructuración de Objetos

Todas las propiedades originales se pasan por valor pero el array es un tipo de dato complejo, y como tal, se para como referencia.

```
console.log(user.features); // ["learn", "code", "paint"]
console.log(fullUser.features); // ["learn", "code", "paint"]
fullUser.features[0] = "program";
console.log(fullUser.features); // ["program", "code", "paint"]
console.log(user.features); // ["program", "code", "paint"]
```

Al cambiar el primer elemento del array features del objeto fullUser, si comprobamos el contenido del objeto user, comprobaremos que también ha cambiado.

Para solucionar esto, podemos hacer lo siguiente:

```
const user = {
        name: "Mario",
        role: "streamer",
        life: 99,
        features: ["learn", "code", "paint"]
}
const fullUser = { ...structuredClone(user), power: 25, life: 50 }
```

Con structuredClone() devolvemos una copia del objeto user, así devolvemos un nuevo objeto, **y no la referencia**.

Desestructuración de Objetos

Estructuras anidadas

```
const user = {
        name: "Mario",
        role: "streamer",
        attributes: {
                height: 183,
                favColor: "blueviolet",
                hairColor: "black"
                }
        }
// Extraemos propiedad attributes (objeto con 3 propiedades)
const { attributes } = user;
console.log(attributes);
// Extraemos propiedad height (number):
const { attributes: { height } } =user;
console.log(height);
// Extraemos height (number) y la cambiamos por nombre size:
 const { attributes: { height: size } } = user; console.log(size);
```

Desestructuración de Objetos

Desestructuración (rest)

Igual que con los arrays, a los objetos se puede aplicar la operación **rest**:

```
const user = {
        name: "Mario",
        role: "streamer",
        life: 99
}
const { name, ...rest } = user;
```

La propiedad name la desestructuramos como variable y rest lo desestructuramos como un objeto que contiene las propiedades role y life.

Desestructuración de Objetos

Parámetros desestructurados

La **desestructuración de parámetros** permite simplificar código.

Ejemplo:

```
const user = {
        name: "Mario", role: "streamer", life: 99
}
function show(data) {
        return `Nombre: ${data.name} (${data.role} y ${data.life} de vida)`;
}
show(user); // "Nombre: Mario (streamer y 99 de vida)"
```

Si desestructuramos los parámetros es más fácil de escribir:

```
const user = {
        name: "Mario", role: "streamer", life: 99
}
function show({ name, role, life }) {
        return `Nombre: ${name} (${role} y ${life} de vida)`;
        }
show(user); // "Nombre: Mario (streamer y 99 de vida)"
```

Si lo necesitasemos, también podríamos usar **rest** en este caso.

Iteración de Objetos

Iteradores de objetos

Los métodos Object.keys(), Object.values() y Object.entries() nos van a permitir realizar esta tarea. Son métodos de una Clase estática, por lo que hay que escribir siempre Object.

- Object.keys(): devuelve un array con las propiedades del objeto
- Object.values(): devuelve un array con los valores de las propiedades del objeto
- Object.entries(): decuelve un array con los pares [key,valor]

```
const user = {
        name: "Mario",
        life: 99,
        power: 10,
        talk: function() {
                return "Hola!";
        }
};
Object.keys(user); // ["name", "life", "power", "talk"]
Object.values(user); // ["Mario", 99, 10, ƒ]
Object.entries(user); // [["name", "Mario"], ["life", 99], ["power", 10], ["talk", ƒ]]
```

Iteración de Objetos

Como un Array también es un Object podemos utilizar estos métodos también para recorrerlos, sólo que en este caso los índices del array son las posiciones (*0, 1, 2, 3...*).

```
const animals = ["Gato", "Perro", "Burro", "Gallo", "Ratón"];
Object.keys(animals); // [0, 1, 2, 3, 4]
Object.values(animals); // ["Gato", "Perro", "Burro", "Gallo", "Ratón"]
Object.entries(animals); // [[0, "Gato"], [1, "Perro"], [2, "Burro"], [3, "Gallo"], [4, "Ratón"]]
```

Iteración de Objetos

Convertir un array a objeto

Usando el método Object.fromEntries(), partiendo de dos arrays, keys y values, donde el primero tiene la lista de propiedades y el segundo tiene la lista de valores.

```
const keys = ["name", "life", "power", "talk"];
const values = ["Mario", 99, 10, function() { return "Hola" }];
const entries = [];
for (let i of Object.keys(keys)) {
        const key = keys[i];
        const value = values[i];
        entries.push([key, value]);
}
const user = Object.fromEntries(entries); // {name: 'Mario', life: 99, power: 10, talk: ƒ}
```

Otra forma, más compacta, con map, sería:

```
const keys = ["name", "life", "power", "talk"];
const values = ["Mario", 99, 10, function() { return "Hola" }];
const entries = values.map((value, index) => [keys[index], value]);
const user = Object.fromEntries(entries);
```

Iteración de Objetos

Recorridos

Se puede usar for..in para recorrer las propiedades de un objeto

```
let unMiembro=new Miembro("Juan",20,"activo");
for (elemento in unMiembro){
    console.log(elemento);
}
```

```
nombre
edad
estado
```

OJO!! Cuando itere, también lo hará con las propiedades que herede.

```
let unProfesor=new Profesor("Juan",20,"activo“,25);
for (elemento in unProfesor){
    console.log(elemento);
}
```

```
nombre
edad
estado
nAlumnos
```

Iteración de Objetos

Recorrer Arrays de Objetos

Se puede definir un array de objetos, y recorrerlo con la instrucción for…in

```
var equipo = [
  {nombre: "David Gómez", numero: "5"},
  {nombre: "Sergio Callejas", numero: "4"},
  {nombre: "Javier Rodríguez", numero: "3"},
  {nombre: "Laura Fernández", numero: "2"},
  {nombre: "Manuel Fernández", numero: "1"}
]

//  recorremos por posición que ocupa en el array
for (let indice in equipo){
   alert(equipo[indice].nombre);
}
```

Como se vió anteriormente, los arrays se pueden recorrer con ***for…of***, que en lugar de usar un índice, recorre directamente los objetos que contiene

Iteración de Objetos

Objetos anidados

Se pueden definir objetos, con propiedades de tipo objeto y métodos.

```
 var factura = {
      empresa: {
        nombre:"CAPALSA",
        direccion: "Poligono Campollano, C",
      },
      cliente: {
        nombre: "IES Leonardo da Vinci",
        direccion: "C/ La Paz s/n",
      },
    elementos: [
        { descripcion: "Folios", cantidad: 100, precio: 5},
        { descripcion: "Toner", cantidad: 20, precio: 30},
      ],
calculaTotal(){
         …….
        }
}
```

Se puede acceder a las propiedades de los objetos:

factura.cliente.nombre