Programación Orientada a Objetos

Contenido

Introducción

Los objetos siempre han estado presentes en el lenguaje JavaScript, con el paso de los años su definición se ha ido modernizando y actualmente disponemos de una más moderna.

- **Objeto JSON:** Rápido y fácil para crear un único objeto. No permite crear múltiples instancias con la misma estructura sin repetir el código.
- **Función Constructora:** Forma más antigua de crear múltiples objetos similares. Usa la palabra clave "**new**" para crear instancias.
- **Clases de ES6:** Sintaxis más moderna y clara que encapsula tanto la estructura como el comportamiento del objeto. Es la forma recomendada para trabajar con POO en JavaScript.

Todas las sintaxis se basan en prototipos y son equivalentes. Y aunque pienses en utilizar únicamente una de ellas lo más seguro es que tus programas acaben siendo una combinación de varias por lo que debes conocerlas todas.

Objeto JSON

En este caso a partir de un objeto JSON podemos definirle métodos y propiedades de sean otros objetos. Esto es cómodo porque como ya hemos dicho JSON nos da facilidades para serializar, almacenar y transmitir información.

El principal problema de esta técnica es que cada objeto tiene su propia estructura lo que nos obliga a repetir código.

```
//Definición mediante objeto JSON
const persona = {
    nombre: 'Juan',
    edad: 30,
    hobbies: ['leer', 'correr', 'programar'],
    direccion: {
        calle: 'Calle Falsa 123',
        ciudad: 'Madrid'
    },
    saludar: function () {
        console.log(`Hola, mi nombre es ${this.nombre}.`);
        console.log(`Vivo en ${this.direccion?.ciudad || 'una ciudad
desconocida'}.`);
    }
};

persona.saludar();
// Salida:
// Hola, mi nombre es Juan.
// Vivo en Madrid.
```

Te habrás fijado en el uso de la propiedad dirección con **“?”**. **Esto nos permite emplear la propiedad aunque no haya sido inicializada evitándose el error**. Llegado el caso, en vez de un error obtenemos un “**undefined**”.

RECUERDA: Al ser un JSON podemos recorrerlo empleando el bucle **“for .. in”** y acceder a las propiedades de un objeto empleando tanto los corchetes **“[]”** como el punto **“.”**.

Otras maneras de obtener información de un objeto pasan por emplear las funciones estáticas del objeto “**Object**”:

- **Object.keys(obj)**: devuelve los nombres de las propiedades del objeto.
- **Object.values(obj)**: devuelve los valores de las propiedades.
- **Object.entries(obj)**: devuelve los pares de propiedad-valor.

Función Constructora

En JavaScript, antes de la introducción de la sintaxis **“class”** en ES6, se utilizaban las funciones constructoras para crear múltiples objetos con la misma estructura y métodos. Una función constructora es simplemente una función especial que, al ser invocada con la palabra clave **“new”**, permite inicializar un nuevo objeto. Dentro de la función constructora, **“this”** hace referencia al objeto que se está creando, y es en este contexto donde se definen las propiedades y métodos del nuevo objeto.

Por ejemplo.

```
//Definición mediante función constructora
function Persona(nombre, edad, hobbies, direccion) {
    this.nombre = nombre;
    this.edad = edad;
    this.hobbies = hobbies;
    this.direccion = direccion;
    this.saludar = function () {
        console.log(`Hola, mi nombre es ${this.nombre}.`);
        console.log(`Vivo en ${this.direccion?.ciudad || 'una ciudad
desconocida'}.`);
    };
}

const persona1 = new Persona(
    'Juan',
    30,
    ['leer', 'correr', 'programar'],
    { calle: 'Calle Falsa 123', ciudad: 'Madrid' }
);

persona1.saludar();
// Salida:
// Hola, mi nombre es Juan.
// Vivo en Madrid.
```

Propiedad “prototype”

Has visto como las funciones constructoras te permiten definir clases fácilmente siguiendo el patrón de una “**closure**”, podemos definir variables y funciones. Este manera de definir clases tiene un pequeño problema, cada vez que instanciamos un nuevo objeto se copian las propiedades y la estructura de las funciones en el nuevo objeto con lo que aumentamos el consumo de memoria.

Todas las funciones en JavaScript (especialmente útil en las funciones constructoras) tienen una propiedad **“prototype”** la cual permite definir métodos y propiedades compartidos entre todas las instancias creadas por la función.

NOTA: el uso de **“prototype”** para definir propiedades no tiene mucha utilidad ya que todas las instancias tendrán el mismo valor, y para esto ya están las constantes.

Por ejemplo.

```
function Persona(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;

    // Método definido internamente (dentro del constructor)
    this.despedir = function () {
        console.log(`Adiós desde ${this.nombre}`);
    };
}

// Método definido en el prototype
Persona.prototype.saludar = function () {
    console.log(`Hola, mi nombre es ${this.nombre}`);
};

const persona1 = new Persona("Juan", 30);
const persona2 = new Persona("Ana", 25);

// Comparación de los métodos entre instancias
console.log(persona1.saludar === persona2.saludar); // true (comparten el
mismo método)
console.log(persona1.despedir === persona2.despedir); // false (cada
instancia tiene su propia copia)
```

De lo anterior se deduce que lo correcto es definir la función constructora como constructor de propiedades y seguido añadirle a través de **“prototype”** todos los métodos de la clase.

Clases de ES6

En ECMAScript 2015 se añadió una mejora de sintaxis para la definición de clases, ten en cuenta que seguimos trabajando sobre prototipos. Ahora las clases de JavaScript proveen una sintaxis mucho más clara y simple para crear objetos y lidiar con la herencia.

RECUERDA: la definición de las clases se realiza en modo estricto, aunque no haya sido indicado.

Por ejemplo.

```
//Definición mediante clase ES6
class PersonaES6 {
    constructor(nombre, edad, hobbies, direccion) {
        this.nombre = nombre;
        this.edad = edad;
        this.hobbies = hobbies;
        this.direccion = direccion;
    }

    saludar() {
        console.log(`Hola, mi nombre es ${this.nombre}.`);
        console.log(`Vivo en ${this.direccion?.ciudad || 'una ciudad
desconocida'}.`);
    }
}

const persona2 = new PersonaES6(
    'Juan',
    30,
    ['leer', 'correr', 'programar'],
    { calle: 'Calle Falsa 123', ciudad: 'Madrid' }
);

persona2.saludar();
// Salida:
// Hola, mi nombre es Juan.
// Vivo en Madrid.
```

A continuación, veremos en detalle las posibilidades de la nueva sintaxis.

Constructor

El constructor se encarga de inicializar la instancia de la clase, consideraciones especiales:

- **Las propiedades privadas deben definirse antes del constructor**.
- Las propiedades públicas pueden definirse en el constructor (cómo en las funciones constructoras) o al mismo nivel que las privadas.

Por ejemplo.

```
class Persona {
    #nombre; // Propiedad privada

    constructor(nombre, edad) {
        this.#nombre = nombre;
        this.edad = edad;
    }

    saludar() {
        console.log(`Hola, mi nombre es ${this.#nombre}`);
    }
}

const persona1 = new Persona("Juan", 30);
console.log(persona1.#nombre); // Error: propiedad privada
```

Fíjate que hemos introducido elementos privados con el prefijo **“#”**. Un elemento privado no es accesible desde fuera de la clase. Más adelante volveremos a este concepto.

Propiedades Getter y Setter

Los **“getters”** y **“setters”** permiten controlar el acceso y la manipulación de las propiedades de la clase. Esto es útil para añadir validación o lógica adicional al leer o escribir una propiedad.

NOTA: para amantes de Java, fíjate que, aunque se definen como función se usan como variable.

Por ejemplo.

```
class Persona {
    constructor(nombre, edad) {
        this._nombre = nombre; // Notación común para propiedades
"privadas"
        this.edad = edad;
    }

    // Getter
    get nombre() {
        return this._nombre;
    }

    // Setter
    set nombre(nuevoNombre) {
        if (nuevoNombre) {
            this._nombre = nuevoNombre;
        } else {
            console.log("Nombre no válido");
```

```
        }
    }
}

const persona1 = new Persona("Juan", 30);
console.log(persona1.nombre); // Juan
persona1.nombre = "Ana";
console.log(persona1.nombre); // Ana
```

Herencia

Podemos hacer que una clase extienda de otra a través de la palabra clave **“extends”**. La clase hija puede hacer referencia a la clase padre a través de la palabra clave **“super”**, esto es útil en los constructores y algunas veces al sobrescribir un método.

En JavaScript no se soporta la herencia múltiple por lo que únicamente podremos tener una clase padre. Dicho lo anterior, se pueden hacen pirulas a través de plantillas de clases mediante la técnica conocida como **“mix-ins”**.

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

    saludar() {
        console.log(`Hola, mi nombre es ${this.nombre}`);
    }
}

class Estudiante extends Persona {
    constructor(nombre, curso) {
        super(nombre); // Llama al constructor de `Persona`
        this.curso = curso;
    }

    estudiar() {
        console.log(`${this.nombre} está estudiando ${this.curso}`);
    }
}

const estudiante1 = new Estudiante("Ana", "Matemáticas");
estudiante1.saludar(); // Hola, mi nombre es Ana
estudiante1.estudiar(); // Ana está estudiando Matemáticas
```

NOTA: en funciones constructoras la herencia se define a través de la propiedad **“__proto__”**.

Elementos privados

Hemo visto que el prefijo **“#”** hace privado el elemento sobre el que aplica. Esto es útil para encapsular la información y el código de nuestros objetos. Podemos definir como elementos privados:

- Propiedades
- Getter y Setter
- Métodos

Por ejemplo.

```
class Persona {
    #_nombre; // Propiedad privada de uso interno "_"


    constructor(nombre) {
        this.#_nombre = nombre;
    }

    // Getter y Setter privados
    get #nombre() {
        return this.#_nombre;
    }

    set #nombre(nuevoNombre) {
        if (nuevoNombre !== this.#_nombre) {
            this.#_nombre = nuevoNombre;
        }
    }

    // Método público
    saludar() {
        this.#nombre = 'Manolo';
        console.log(this.#obtenerSaludo());
    }

    // Método privado
    #obtenerSaludo() {
        return `Hola, mi nombre es ${this.#nombre}`;
    }
}

const persona1 = new Persona("Juan");
persona1.saludar(); // Hola, mi nombre es Manolo
//persona1.#obtenerSaludo(); // Error: método privado
```

Métodos y propiedades estáticos

Los métodos y propiedades estáticas son elementos que se asocian a la clase en sí misma, en lugar de a las instancias de la clase. Se definen mediante la palabra clave "`static`".

La utilidad de los métodos y valores estáticos es la de crear funcionalidades de clase o constantes de clase.

Por ejemplo.

```
class Persona {
    // Propiedad estática
    static especie = "Oompa Loompa de la informática";

    // Constructor
    constructor(nombre = "Desconocido", edad = 18) {
        this.nombre = nombre;
        this.edad = edad;
    }

    // Método normal
    saludar() {
        console.log(`Hola, mi nombre es ${this.nombre}`);
    }

    // Método estático
    static crearAnonimo() {
        return new Persona("Anónimo", 0);
    }
}

const persona1 = new Persona("Juan", 30);
persona1.saludar(); // Hola, mi nombre es Juan

// Accediendo a la propiedad estática
console.log(Persona.especie); // Oompa Loompa

const personaAnonima = Persona.crearAnonimo();
personaAnonima.saludar(); // Hola, mi nombre es Anónimo
```

Comportamiento de "this", funciones normales y funciones arrow

Vamos a tratar de aclarar el comportamiento de "this" ya que puede dar origen a errores.

Por defecto, **"this" hace referencia al contexto en el que se ejecuta la función**. Ese contexto es el objeto que posee la función. Por ejemplo, si defines una función dentro de un objeto, "this" dentro de esa función apuntará al objeto.

“this” en funciones normales

Si llamas a la función desde un objeto, “this” apunta a ese objeto.

Si llamas a **“this”** desde el contexto global, **“this”** apunta a **“window”** (en JS de navegador) o es **“undefined”** en modo estricto.

En resumen, en este caso el valor de **“this”** depende de cómo y desde dónde se llama a la función.

“this” en funciones Arrow

Las funciones arrow (=>) TIENEN SU PROPIO **“this”**. Por defecto NO HAY UN NUEVO **“this”** PARA CADA OBJETO SINO QUE SE HEREDA DEL ENTORNO EN QUE SE DEFINE.

En resumen, en este caso el valor de **“this”** se hereda del entorno de definición de la función.

Prueba el siguiente ejemplo, **EL MISMO CÓDIGO GENERA DOS SALIDAS DISTINTAS**.

```
'use strict';

// No hay contructor, persona se define en global, la arrow está en
global
const persona = {
    nombre: "Juan",
    saludarArrow: () => {
        console.log(`ARROW: Hola, mi nombre es ${this.nombre}`);
    },

    saludarTradicional: function () {
        console.log(`TRADICIONAL: Hola, mi nombre es ${this.nombre}`);
    }
};

persona.saludarArrow();  // Salida: ARROW: Hola, mi nombre es undefined
persona.saludarTradicional(); // Salida: TRADICIONAL: Hola, mi nombre es
Juan

// El objeto tiene un nuevo contexto propio del objeto distinto del
global, la arrow apunta al objeto Persona
class Persona {
    constructor(nombre) {
        this.nombre = nombre;
    }

    saludarTradicional() {
        console.log(`TRADICIONAL: Hola, mi nombre es ${this.nombre}`);
    }
```

```
    saludarArrow = () => {
        console.log(`ARROW: Hola, mi nombre es ${this.nombre}`);
    }
}

const persona2 = new Persona("Ana");

persona2.saludarArrow(); // Salida: ARROW: Hola, mi nombre es Ana
persona2.saludarTradicional(); //Salida: Tradicional: Hola, mi nombre es
Ana

console.log('Ponemos el temporizador para forzar un callback');
// Usamos setTimeout para simular un callback
// FIJATE QUE EL OBJETO PIERDE SU "this" AL EJECUTARSE EN OTRO CONTEXTO
// El arrow lo conserva porque se hereda
setTimeout(persona2.saludarTradicional, 1000); // Salida: TRADICIONAL:
Hola, mi nombre es undefined
setTimeout(persona2.saludarArrow, 1000);       // Salida: ARROW: Hola, mi
nombre es Ana
```

Reglas generales de uso funciones normales y funciones arrow

Dependiendo del caso de uso deberás emplear un tipo de función u otro.

- La opción recomendada son las “funciones normales”.
- Las “funciones arrow” sólo son útiles cuando necesitemos ejecutar el código en “callbacks”.

Los puntos para justificar lo anterior son los siguientes:

- Las funciones arrow no se pueden sobrescribir por las clases hijas.
- Las funciones arrow no tienen acceso a “super”.
- Las funciones arrow se definen para cada instancia de una clase, no van en el prototipo, por lo que consumen más memoria y degradan el rendimiento.

Módulos JavaScript

La modularidad es una práctica de diseño que permite dividir una aplicación en componentes independientes llamados "módulos". Cada módulo contiene partes específicas de la funcionalidad de la aplicación, lo que facilita la organización y la reutilización del código. La modularidad es clave para proyectos grandes, ya que mejora la mantenibilidad y facilita el trabajo en equipo al permitir que diferentes desarrolladores trabajen en módulos independientes.

Antes de ES6, JavaScript carecía de un sistema de módulos nativo y se empleaban soluciones de terceros. Con ES6, JavaScript introdujo su propio sistema de módulos, estandarizando cómo los desarrolladores pueden dividir y reutilizar el código.

La estructura modular de ES6 se basa en dos palabras clave:

- **“export”**: Permite marcar las funciones, variables y clases que queremos hacer públicas y accesibles desde otros módulos.
- **“import”:** Facilita el uso de funcionalidades definidas en otros módulos, ofreciendo opciones específicas y selectivas de importación.

Adicionalmente podemos definir un ÚNICO ELEMENTO **“default”** (por fichero) para indicar cual es elemento por defecto a importar del módulo, es opcional y se emplea por valores semánticos más que por usos prácticos. **RECUERDA**, el “**import**” para elementos “**default**” no va rodeado de **{}**.

Veamos como definir un módulo.

```
// modulo-matematicas.js

// Constante
export const PI = 3.1416;

// Funciones
export function sumar(a, b) {
    return a + b;
}

export function restar(a, b) {
    return a - b;
}

export function multiplicar(a, b) {
    return a * b;
}

// Clase Calculadora como exportación por defecto
export default class Calculadora {
    static suma(a, b) {
        return sumar(a, b);
    }

    static resta(a, b) {
        return restar(a, b);
    }

    static multiplica(a, b) {
        return multiplicar(a, b);
    }
}
```

IMPORTANTE: hay una sintaxis alternativa que a mi parecer resulta más cómoda y común.

```
const PI = 3.1416;
```

```
function sumar(a, b) {…}

function restar(a, b) {…}

function multiplicar(a, b) {…}

class Calculadora {…}


// Exportaciones al final del archivo
export default Calculadora;
export { PI, sumar, restar, multiplicar };
```

Para usar el módulo debemos importarlo.

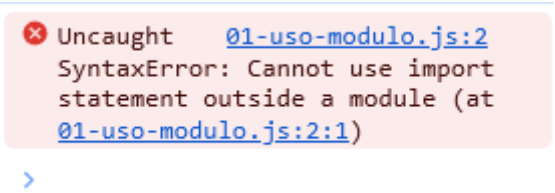
```
// importo elementos de modulo-matematicas.js
import Calculadora, { PI, sumar, restar, multiplicar } from './modulo-
matematicas.js';

console.log("PI:", PI); // PI: 3.1416
console.log("Suma:", sumar(4, 5)); // Suma: 9
console.log("Resta:", restar(10, 7)); // Resta: 3
console.log("Multiplicación:", multiplicar(3, 3)); // Multiplicación: 9

// Uso de la clase Calculadora
console.log("Calculadora Suma:", Calculadora.suma(10, 20)); //
Calculadora Suma: 30
console.log("Calculadora Multiplicación:", Calculadora.multiplica(4, 4));
// Calculadora Multiplicación: 16
```

FIJATE, que al importar rodeamos con llaves “{}” a los elementos nombrados ordinarios que deseamos importar. En el caso de la opción por defecto las llaves no son necesarias.

Al ejecutar el código anterior tendremos el siguiente error.



Para poder usar un módulo y que no se produzca el error debemos adaptar la sintaxis del fichero HTML que lo usa. Añadimos `type="module"` a la etiqueta **“script”**.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
```

```
    <title>Módulos JavaScript</title>
    <script type="module" src="04-Modulos/01-uso-modulo.js"></script>
</head>
<body>

</body>
</html>
```

Uso de alias

Se pueden renombrar elementos importados con “as”, esto puede ser especialmente útil cuando tengamos varios módulo y se produzcan conflictos con los nombres repetidos.

Tenemos 3 opciones:

- **Alias en exportaciones nombradas**. Sintaxis “**{ nombreOriginal as alias }**”

Por ejemplo.

```
import { PI as numeroPi, sumar as add, restar as subtract } from
'./modulo-matematicas.js';
console.log("Suma:", add(4, 5)); // Suma: 9
```

- **Alias en exportación por defecto**. Sintaxis “**import aliasEleDefecto from ‘módulo’ }**”

Por ejemplo.

```
import CalculadoraMatematica from './modulo-matematicas.js';
console.log("Calculadora Suma:", CalculadoraMatematica.suma(10, 20));
// Calculadora Suma: 30
```

- **Alias par importar todo el módulo**. Sintaxis “**import * as aliasModulo from ‘módulo’ }**”

Por ejemplo.

```
import * as Matematicas from './modulo-matematicas.js';
console.log("Suma:", Matematicas.sumar(4, 5)); // Suma: 9
```

Consideraciones del empleo de módulos

Algunas cosas que debes recordar al trabajar con módulos:

- Los módulos **siempre se ejecutan en modo estricto** (strict mode).
- **Cada módulo tiene su propio ámbito local**, no mezclando sus definiciones con el ámbito global ni con el código de otros módulos.
- Los módulos se cargan automáticamente de manera asíncrona (async) y diferida (defer). No siendo necesario indicar estas propiedades.

Bibliografía

Documentación MDN objetos en JavaScript

https://developer.mozilla.org/es/docs/Learn/JavaScript/Objects/Basics

Documentación MDN class

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Classes

Documentación MDN export

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/export

Documentación MDN import

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/import

Manual JavaScript.info Objetos básicos

https://es.javascript.info/object-basics

Manual JavaScript.info Class

https://es.javascript.info/class