UD2 – POO y módulos

DAW2 - DWEC

POO con JavaScript

3 formatos.

- Objetos JSON
- Funciones constructoras
- Clases ES6

Todos se basan en “Prototipos” => plantillas de código

Trabajando instancias

for .. in: nos devuelve cada propiedad del objeto.

[propiedad] y .propiedad: nos permite acceder a la propiedad.

“.propiedad?”: devuelve false si la propiedad no está definida, evita error de lectura.

=> `${this.direccion?.ciudad`

Objeto “Object”

- Object.keys(obj)
- Object.values(obj)
- Object.entries(obj): lista de pares clave-valor-

Objetos JSON

Tienen el formato JSON con las ventajas e inconvenientes que ello aporta.

PROBLEMA, cada instancia tiene una copia de la estructura del objeto.

```
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'}.`);
    }
};
```

Función Constructora

Función que inicializa una instancia, se comporta como el constructor de la clase.

Los métodos se definen como propiedades a las que se asigna una función.

Para crear una nueva instancia hay que usar el “new”.

```
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'}.`);
    };
}
```

Propiedad “prototype”

El problema de las funciones constructoras es que al definir un método el código se copia en cada instancia.

“prototype” permite compartir el mismo código entre todas las instancias de la clase.

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

    this.despedir = function () {
        console.log(`Adiós desde ${this.nombre}`);
    };
}

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

Formato ES6 - Class

Permite aplicar un formato moderno a la definición de clases.

El código de las clases usa modo estricto.

Disponemos de:

- Constructor
- Getter y Setter
- Herencia
- Elementos estáticos “static”.
- Elementos privados

```
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'}.`);
    }
}
```

Getter y Setter, elementos privados

Disponemos de Getter y Setter, se definen como funciones, pero se usan como variables.

Podemos hacer privado un elemento con “#”

Prefijo “_”, indica uso interno (por convenio)

```
class Persona {
    #_nombre;

    get nombre() {
        return this.#_nombre;
    }

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

    saludar() {}
    #obtenerSaludo() {}
}

const persona1 = new Persona("Juan");
console.log(persona1.nombre);
```

Herencia

No hay soporte para herencia múltiple.

Heredamos con “extend”.

Invocamos al objeto padre con “super”.

```
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}`);
    }
}
```

Variable “this”

En POO deseamos que “this” haga referencia a la instancia actual.

En JS según la definición y el contexto podemos perder la referencia a “this”.

REGLAS DE USO

Usar siembre funciones nombradas para código común.

Usar funciones Arrow siempre que se usen en callback.

Módulos

Los módulos nos permiten dividir una aplicación grande en piezas más pequeñas.

Facilitan la organización y reutilización del código.

Un fichero JS que importe un módulo debe emplear la propiedad `type="module"`

```
<script type="module" src="04-Modulos/01-uso-modulo.js"></script>
```

Consideraciones.

- Se ejecutan en modo estricto.
- Cada módulo tiene su propio ámbito local.
- Se cargan automáticamente de manera asíncrona y diferida.

Módulos - Export

Un fichero que exporte definiciones se convierte en módulo.

Para exportar usamos “export”.

Podemos exportar elementos uno a uno o en conjunto al final del archivo.

Podemos definir un elemento por defecto “default”.

```
export default Calculadora;
export { PI, sumar, restar, multiplicar };
```

Módulos - Import

Para importar usamos “import”.

- El elemento por defecto no se envuelve con {}
- Los elementos comunes se envuelven con {}

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

Podemos importar algunas definiciones o todas.

Podemos emplear alias al importar “as” o ponerle un alias a todo el módulo.

```
import { sumar as add, restar as subtract } from ...
import * as Matematicas from ...
```

Preguntas