4.- ESTRUCTURAS DEFINIDAS POR EL USUAIRO EN JAVASCRIPT.

DWEC
Mª Luz Sánchez Rubio

OBJETIVOS

- Poder crear funciones personalizadas para realizar tareas específicas que las funciones predefinas no logran hacer.
- Comprender el objeto Array de JavaScript y familiarizarse con sus propiedades y métodos.
- Crear objetos personalizados diferentes a los objetos predefinidos del lenguaje.
- Definir propiedades y métodos de los objetos personalizados.
- Uso de módulos
- Conocer y aplicar los patrones de diseño en JavaScript

1.- Funciones

Las funciones son líneas de código que realizan una serie de acciones y vienen a resolver uno de los principales problemas de la programación estructurada.

Ventajas:

- Permiten reutilizar código, mejorar la eficiencia, aumentar su legibilidad y reducir los costes de mantenimiento de los programas.

Ejemplo:

- Usar una función que implemente la validación de un formulario y utilizarla en todos los formularios que se programen.

1.- Funciones

1.1.- Declaración de función y expresión de función

- **Declaración de función**

```
function nombreFuncion ( [params]){
        <cuerpo función>
}

nombreFuncion (); //llamada a la función
```

- **Ejemplo// Crear una función que reste dos valores**

```
function restar ( ){
   console.log(6-4);
}

restar (); //llamada a la función
```

1.- Funciones

1.2.- Declaración de función y Expresión de función

- **Expresión de función**

```
const/let nombreFuncion=function ( [params]){
        <cuerpo función>
}

nombreFuncion();  //llamada a la función
```

- **Ejemplo// Crear una función que reste dos valores**

```
const restar =function( ){
    console.log(6-4);
}

restar();  //llamada a la función
```

1.- Funciones

1.3.- Parámetros y Argumentos en funciones

- **Los parámetros son la lista de variables que se indican al definir una función. Por ejemplo, a y b**

```
const/let nombre=function (a, b){
        <cuerpo función>
}
```

- **Los argumentos son los valores que se pasan a la función cuando se invoca. Por ejemplo, 6 y 4**

```
restar(6, 4);  //llamada a la función
```

1.- Funciones

1.4.- Parámetros por defecto

- **Los parámetros por defecto en la función permiten que los parámetros con nombre se inicien con valores predeterminados si no se pasa ningún valor o undefined**

```
function multiplicar (a, b=1){
    console.log(a*b)
    }

multiplicar(5); //muestra 5
multiplicar(5,2), //muestra 10
```

1.- Funciones

1.5.- return

La sentencia return finaliza la ejecución de la función y especifica un valor para ser devuelto a quien llama a la función.

```
const multiplicar=function(a,b){
        return (a*b)
}
```

1.- Funciones

1.6.- Ámbito |scope de variables

Cómo ya se ha visto en la primera unidad, al hablar de ámbito de variables, se está haciendo referencia a aquellas partes del programa donde una variable o constante es "visible", accesible.

Si una variable es accesible solo desde el interior de una función se dice que la variable es local a la función

```
let mensaje="variable global";
const mostrar=function(){
    let mensaje='variable local';
    console.log(mensaje); // variable local;
}
mostrar();
console.log(mensaje); // variable global;
```

1.- Funciones

1.7.- Funciones flechas o Arrow functions

- Las funciones flechas son una alternativa abreviada a una función tradicional, pero es limitada y no se puede utilizar en todas las situaciones:

  Limitaciones:

  - No tiene sus propios enlaces a this o super.
  - No es apta para los métodos call, apply y bind
  - No se puede utilizar como constructor
  - No se puede utilizar yield dentro de su cuerpo.

1.- Funciones

1.7.- Funciones flechas o Arrow functions

Ejemplos//

```
//función tradicional
const multiplicar=function(a,b){
        return (a*b)
}
```

//1.- función arrow. Elimina la palabra "function" y coloca la flecha entre el parámetro y la llave de apertura.

```
const multiplicar=(a,b)=>{
        return (a*b)
}
```

1.- Funciones

1.7.- Funciones flechas o Arrow functions

Ejemplos//

//2.- función arrow. Quita las llaves y la palabra “return”

```
const multiplicar=(a,b)=> a*b;
```

//3.- función arrow. Suprime los paréntesis si solo hay un parámetro

```
const multiplicar=a=> a*5;
```

1.- Funciones

1.8.- Closures

- Una función **closure** (o clausura en español) es una función en JavaScript que “recuerda” el entorno en el que fue creada.

- Una closure puede acceder a las variables de su ámbito léxico (es decir, las variables declaradas en la función o en el bloque donde la **closure** fue definida, incluso después de que esa función o bloque haya terminado de ejecutarse.

1.- Funciones

1.8.- Closures

Aplicaciones comunes de las closures:

- ✓ **Contadores y acumuladores:** Creación de contadores y acumuladores privados.
- ✓ **Funciones de fábrica:** Creación de funciones personalizadas basadas en parámetros específicos.
- ✓ **Módulos:** Implementación de patrones de diseño como el módulo en JavaScript, donde se encapsulan datos y funciones en un ámbito privado

1.- Funciones

1.8.- Closures

Ejemplo://

```
function createCounter() {
   let count = 0; // Variable local que puede ser accesible por la closure

   // Closure
   return function () {
      count += 1;
      return count;
    };
}

const counter1 = createCounter();
console.log(counter1()); // 1
console.log(counter1()); // 2

const counter2 = createCounter();
console.log(counter2()); // 1
console.log(counter2()); // 2
```

2.- Arrays

2.1.- Creación

Un array es una colección o agrupación de elementos en una misma variable, cada uno de ellos ubicado por la posición que ocupa en el array.

Definiciones//

const letras= new Array (“a”, “b”, “c”); //Array de 3 elementos

const letras =new Array(3) //Array vacío de tamaño 3

//declaración apropiada

const letras=[“a”, “b”, “c”]; //Array de 3 elementos

const letras=[]; //Array vacío

const letras=[“a”, 5, true]; //Array mixto

3.- Conjuntos

3.1.- Creación de un conjunto

Los conjuntos o sets, son estructuras de datos muy parecidas a los arrays pero con la particularidad de que no permiten valores duplicados.

```
const conjunto = new Set();
```

Para indicarle, desde su declaración, los elementos que lo componen inicialmente, es preciso pasarle un objeto de tipo iterable (array, map, string, set)

```
const set= new Set([1, "a", true]);
```

4.- Mapas

4.1.- Creación de un mapa

Los mapas son estructuras de datos muy parecidas a los arrays pero con la particularidad que su estructura tiene valores guardados a través de una clave para identificarlos. Comúnmente, se denomina **pares clave-valor**

```
const  map = new Map();
```

```
const  map = new Map([1, “uno”], [2, “dos”]);
```

Para indicarle, desde su declaración, los elementos que lo componen inicialmente, es preciso pasarle un objeto de tipo iterable (array, map, string, set)

```
const set= new Set([1, “a”, true]);
```

5.- POO en JavaScript

5.1.- Introducción

La POO formalmente, necesita que el modelo de programación esté basado en clases, pero JavaScript no es un lenguaje basado en clases, sino basado en prototipos.

Toda aplicación JavaScript usa objetos y basa su lógica en las interacciones entre ellos. También es posible crear objetos sin utilizar clases. En su caso, utiliza prototipos que, igualmente, van a permitir heredar propiedades y métodos, y añadir otros específicos.

5.- POO en JavaScript

5.2.- Creación

```
Class Animal {
    // Propiedades
    name= “xxxxx”;
    type= “yyy”;

    //Métodos
    comer() {
      return “Se lo comen todo”;
    }
}
```

6.- Módulos

5.1.- Introducción

A partir de ES2015 se introduce una característica nativa denominada **Módulos ES** que permita la importación y exportación de fragmentos de datos entre diferentes ficheros JavaScript, eliminado las desventajas que habías hasta ahora y permitiendo trabajar de forma más flexible en el script

7.- Patrones de diseño

7.1.- Introducción

Los patrones de diseño son soluciones probadas a problemas típicos y recurrentes que nos podemos encontrar a la hora de desarrollar una aplicación.

Ante un problema que nos podamos encontrar en el desarrollo de nuestra aplicación, es mejor solucionarlo usando un patrón, en lugar de reinventar la rueda.

Algunas ventajas de aplicar patrones de diseño:

- Simplificar los problemas.
- Estructurar mejor el código.
- Hacer el código más predecible

7.- Patrones de diseño

7.2.- Tipos

Clasificación:

- **Patrón de Diseño Singleton**
- **Patrón de Diseño Factory**
- **Patrón de Diseño Module**
- **Patrón de Diseño Observer**

7.- Patrones de diseño

7.3.- Patrón de Diseño Singleton

El patrón de Diseño Singleton se utiliza para asegurarse que una clase solo tenga una instancia y proporciona un punto de acceso global a esa instancia.

Utilidades:

- Cuando se necesita un objeto único para coordinar acciones o recursos compartidos
- Se puede acceder a la instancia única desde cualquier parte del programa. Esto evita la necesidad de pasar la instancia como parámetro o crear múltiples instancias en diferentes partes del código.
- También se puede utilizar para conexiones a bases de datos, archivos de configuración o registros

7.- Patrones de diseño

7.4.- Patrón de Diseño Factory

El patrón de Diseño Factory se utiliza para encapsular la creación de objetos y permitir la creación de diferentes tipos de objetos a través de una interfaz común

Utilidades:

- Encapsulación de la lógica de creación. Esto ayuda a mantener el código más organizado y modular, ya que la creación de objetos se centraliza en un solo lugar.
- Abstracción de la creación de objetos. Significa que el código que utiliza los objetos no necesita conocer los detalles de cómo se crean.
- Flexibilidad en la creación de objetos. Permite crear diferentes tipos de objetos utilizando un interfaz común.
- Separación de responsabilidades. Se separa la responsabilidad de la creación de objetos de la lógica del negocio, ayudando a mantener un código más limpio y facilita el mantenimiento y la evolución del sistema.

7.- Patrones de diseño

7.5.- Patrón de Diseño Module

El patrón de Diseño Module es un patrón estructural que se utiliza principalmente para encapsular un consjunto de funciones, variables y objetos relacionados dentro de una unidad

Utilidades:

- Permite ocultar detalles internos y exponer solo lo necesario.
- Ayuda a organizar el código y a evitar la contaminación del espacio de nombres globales.
- Promueve la separación de responsabilidades y el código mantenible.

7.- Patrones de diseño

7.6.- Patrón de Diseño Observer

El patrón de Diseño Observer se utiliza para establecer una relación de uno a muchos objetos, de modo que cuando un objeto cambia de estado, notifica y actualiza automáticamente a otros objetos que dependen de él.