CLASE NRO. 15 – FUNCIONES. ENTIDADES DE PRIMERA CLASE

Antes de introducirnos en las técnicas que disponemos en JavaScript para manejar el SINCRONISMO / ASINCRONISMO y como estas están directamente vinculadas a funciones, es más las tres técnicas que veremos están directamente vinculadas al concepto de funciones a continuación veremos y repasaremos algunas particularidades que tienen las funciones, observaremos toda la potencia que se puede conseguir con las funciones.

Para ello es importante e imprescindible comprender los conceptos de entidades de primera clase y las particularidades de las mismas.

Las entidades de primera clase en JavaScript son aquellas que pueden:

- ✓ Ser asignadas a una variable ó constante.
- ✓ Ser pasadas como argumentos a funciones.
- ✓ Ser devueltas como resultado de funciones.
- ✓ Ser almacenadas en estructuras de datos.

Las funciones son entidades de primera clase porque se pueden hacer las siguientes operaciones o particularidades.

- **Particularidad 1:** Se pueden declarar de forma clásica

```
function Cuadrado (parametro)
{
        return parametro * parametro;
}
```

- **Particularidad 2:** Se pueden guardar dentro de constantes o variables

```
const Cuadrado = function(parametro)
{
   return (parametro * parametro);
}
```

- **Particularidad 3:** Se pueden declarar como arrow functions y guardarlas en constantes

```
const Cuadrado2 = (parametro)=>{
   return parametro * parametro;
}
```

- **Particularidad 4:** Se pueden declarar como arrow functions y si solo reciben un parámetro y es una sola operación que se realiza con el parámetro no es necesario que lleven () para el parámetro, no necesitan que lleven las {} para el cuerpo de la función y no es necesario que lleve la palabra return.

```
const Cuadrado3 = parametro => parametro * parametro;
```

- **Particularidad 5:** Funciones pueden ser pasadas como argumento ó parámetro de otra función por ende hay dos funciones. una podríamos decirle función llamadora y la otra le podríamos decir función interna.

```
// esta es la función llamadora que recibe como parámetro una función que se llamara interna //

const funcionLlamadora = (fnInterna)=>
{

   // aquí invocamos a la función interna //
   fnInterna();

   // esto sería el código de la función llamada //
   console.log("esto es código de la función Llamadora");
}
```

- **Particularidad 6:** Funciones que retornan otras funciones.

```
const funcionRetornaOtraFuncion = ()=>
        {
           return (parametro)=>
              {
                 return parametro * parametro;
              }
        }
```

- **Particularidad 7:** El resultado de funciones puede ser pasado como parámetro de otras funciones.

```
function Cubo(numero)
  {
     return (numero * numero * numero);
  }


  let ResultadoCuadrado = Cuadrado(5);

  let Resultado2 = Cubo(ResultadoCuadrado); // aqui tendría la potencia 2 de 5;

  console.log(Resultado2);//aqui tendría la potencia 5 de 5
```

- **Particularidad 8:** se pueden establecer valores por defecto en los parámetros de entada de una función.

```
function Promedio(num1=0,num2=0,num3=0)
        {
           return (num1 + num2 + num3)/3;
        }
```

- **Particularidad 9:** pueden ser almacenadas y ser parte de cualquier estructura de datos.

```
class CalculosEstadisticos
    {
      constructor(num1,num2)
      {
        this.Numero1 = num1;
        this.Numero2 = num2;
      }

      // es una función que está dentro de una clase = estructura.
      //pero aquí por convención
      // les decimos que se llaman métodos.
      devolverMediaAritmetica()
      {
        return (this.Numero1 + this.Numero2 )/2;
      }
    }
```

- **Particularidad 10:** funciones pueden recibir como parámetro otras funciones, que pueden ser invocadas dentro de la función llamadora podríamos decir, podríamos definir función parámetro a la función que se pasa como parámetro y función llamadora o (callback) a la función que llama a la primera. Entonces quedaría algo así.

```
  // esta es una función principal que llama a otras dos //
const fnLLamadoraOCallBack = function(f1,f2)
{
  f1();
  f2();
}

// en la invocación de la función prinicipal le paso dos funciones
// que se acaban de crear y se envían en el parámetro

fnLLamadoraOCallBack(()=>{
  console.log("esto es el cuerpo de la primera callback");
},
(

)=>{
  console.log("este es el cuerpo de la segunda callback");
});
```

Particularidades a modo de comentario:

Particularidad 11: Funciones Anónimas Autoejecutables (IIFE): Estas funciones se ejecutan inmediatamente después de ser definidas.

```
(function() {
        console.log('IIFE ejecutada');
})();
```

Particularidad 12: Funciones Generadoras: Las funciones generadoras (function*) permiten pausar y reanudar la ejecución del

```
function* generador() {
        yield 1;
        yield 2;
        yield 3;
}

const gen = generador();
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
console.log(gen.next().value); // 3
```

Particularidad 13: Funciones Asíncronas (async/await): Las funciones asíncronas permiten escribir código asincrónico de manera más sencilla.

```
async function fetchData()
{
        const response = await fetch('https://api.example.com/data');
        const data = await response.json();
        console.log(data);
}

fetchData();
```

Particularidad 14: Closures. Una función puede acceder a las variables de su contexto exterior incluso después de que ese contexto haya terminado.

```
function crearContador() {
  let contador = 0;
  return function() {
    contador++;
    return contador;
  }
```

```
}

const contador = crearContador();
console.log(contador()); // 1
console.log(contador()); // 2
```

Particularidad 15: Funciones con Rest Parameters: Permiten a una función aceptar un número indefinido de argumentos como un array.

```
function sumar(...numeros) {
return numeros.reduce((acc, num) => acc + num, 0);
}

console.log(sumar(1, 2, 3)); // 6
```

Particularidad 16: Funciones con Spread Operator: Permiten expandir elementos de un iterable (como un array) en lugares donde se esperan cero o más argumentos.

```
function sumar(x, y, z) {
        return x + y + z;
}

const numeros = [1, 2, 3];
console.log(sumar(...numeros)); // 6
```

Importancia de las Entidades de Primera Clase:

Tener entidades de primera clase en un lenguaje de programación permite una mayor flexibilidad y modularidad. Se puede crear código más dinámico y reutilizable, y se pueden aplicar patrones de diseño más avanzados.

Por ejemplo, el hecho de que las funciones sean de primera clase en JavaScript permite la programación funcional, donde se pueden crear funciones de orden superior (funciones que toman otras funciones como argumentos o que devuelven funciones). Esto también facilita el uso de callbacks y promesas para manejar la asincronía, lo que es una parte fundamental del desarrollo moderno en JavaScript.