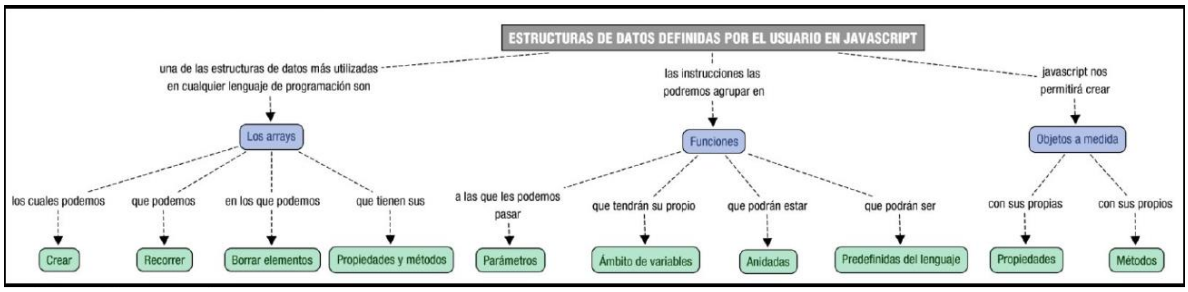
TEMA 4. Programación con funciones, arrays y objetos definidos por el usuario

Índice

Autor: Roberto Marín

Asignatura: Desarrollo Web Entorno Cliente (DAW2)

Año: 2024/2025



4.1. Funciones

- Funciones predefinidas del lenguaje
- Llamadas a funciones. Definición de funciones.

https://www.w3schools.com/js/js_function_definition.asp

https://www.w3schools.com/js/js_function_parameters.asp

https://www.w3schools.com/js/js_function_call.asp

https://www.w3schools.com/js/js_function_apply.asp

https://www.w3schools.com/js/js_function_closures.asp

Las funciones en JavaScript nos van a permitir reutilizar código. Para declarar una función utilizaremos la palabra reservada ***function*** y le daremos un nombre (más adelante veremos que no siempre). A la función le pasaremos una serie de parámetros (opcionalmente) y nos devolverá un resultado, o no. Si la función hace un 'return' en ese momento detendrá su ejecución.

```
function suma(a, b) {
return a + b;
console.log("Esto no aparece");
}
let s = suma(3, 4);
console.log(s);
let x = 10;
let y = 12;
let t = suma(x, y);
console.log(t);
```

Ejercicio:
Cree un script que permita realizar el cálculo de la nomina. La función debe recibir como parámetro el salario bruto anual, la retención a aplicar y el número de pagas.
El resultado devuelto por la función será el salario neto mensual.

JavaScript permite que el número de argumentos pasado a una función sea diferente al que se declara en dicha función.

```
<body>
    <p id="demo1"></p>
</body>
```

```
let x = nomina(15000,18,14);
document.getElementById('demo1').innerHTML = x;

function nomina(salarioBruto, retencion, numeroPagas) {
let salarioNeto= salarioBruto * (1- (retencion/100)) / numeroPagas
return salarioNeto;}
```

Ejercicio: Crea un script con una función y pásale dos argumentos a y b. Comprueba que si b no es un número, lo convertimos en 0 y haremos la suma de a y b. Comprobar resultados.

```
console.log(funcion(2,3));console.log(funcion(2));
```

Pero JavaScript nos da una opción más interesante para este tipo de situaciones con el **operador OR (||)**. Cuando hacemos (foo || bar) el operador OR evalúa si foo es true o cualquier otro valor transformable en true. En ese caso se devuelve el valor de foo.

Ejercicio: Aplica al enunciado anterior la siguiente solución, mucho más elegante:

```
b || (b = 0);
```

Así el parámetro no se sobrescribe de no ser necesario.

Como hemos visto el número de argumentos en las funciones JavaScript es variable y puede ser que se 'pierdan' argumentos o q variables de la función 'lleguen' como undefined. JavaScript nos ofrece una herramienta arguments que recoge todas las variables 'pasadas' a la función. (*Sistemas Informáticos: S* en Bash*). **El objeto arguments** del tipo Arguments se puede asimilar a efectos prácticos a un array y nos permitirá la siguiente ejecución. Para ello emplearemos la propiedad arguments.lenght.

Ejercicio: Imaginemos que queremos calcular la media de dos cadenas de números. La primera con 3 y la segunda con 6 números. Para ello usa una función de JavaScript.

```
Ayuda:
console.log("la media es "+media(5,6,7));
console.log("la media es "+media(5,6,7,7,10,12));
```

Ejercicio:
Defina una función para validar el número de matrícula de un alumno.
Para ello:

- El número de matrícula está compuesto por 5 números y una letra.
- Además el primer número tiene que ser impar
- La letra tiene que ser P ó S para diferenciar los estudios.

Ejercicio:
Crea un script con una función para identificar un número de teléfono fijo.

- Empezará siempre por 8 o 9
- Deberá contener 9 números
- Ninguna negativo

No puede coincidir que haya dos ceros en la segunda y tercera posición

4.1.1 Valores por defecto en parámetros de funciones

A partir de la especificación ECMAScript2015 (ES6) se introduce la posibilidad de usar valores por defecto en las funciones. Los valores por defecto se usarán por la función en el caso de que no se le pase un parámetro o bien este sea undefined Por ejemplo para una función definida como:

```
function sum(a,b=0){…}
```

Los resultados serán:

```
suma(1) // b es 0
suma(1,2) // b es 2
suma(1,undefined) // b es 0
```

Los valores por defecto pueden estar basados en parámetros anteriores, así:

```
function prueba(a , b, r = a - b){
return(r);
}
prueba(3,2); // devuelve 1
prueba(2,3,5); //devuelve 5
```

PROFUNDIZAR ALGO MÁS CON ARGUMENTS. Para aprender más y mejor aquí hay un gran artículo http://www.etnassoft.com/2016/06/28/parametros-obligatorios-y-valores-por-defecto-en-funciones-javascript-con-es6/

4.1.2 Parámetros requeridos

Si ponemos como valor por defecto una función, podremos gestionar así los errores cuando no pasamos los parámetros.

```
let esRequerido = function () {
throw new Error( 'Parámetro no enviado ');
};
let funcionEjemplo = function ( foo = esRequerido() ) {
return foo;
};
console.log( funcionEjemplo( 'Se envía parámetro' ) );
console.log( funcionEjemplo() ); // Error: Parámetro no enviado
```

Otro ejemplo más para que quede más claro:

```
//Parametros requeridos
let esRequerido = function () {
//throw new Error( 'No se pasa numero a la función ');
};
let cuadrado = function ( n = esRequerido() ) {
return n*n; };
console.log( cuadrado ( 3 ) ); // Devuelve 9
console.log( cuadrado () ); // Error: Parámetro no enviado
```

4.1.3 Funciones flecha

Otra de las novedades de ES6 es la sintaxis de flecha para definir las funciones:

```
let suma = (n,m) => n+m;
console.log(suma(3,2)); // 5
```

La misma función en versión clásica sería:

```
    function suma(n,m){
      return n+m;
      }
    console.log(suma(3,2))
```

NOTA: Las funciones flecha son siempre anónimas

Como se ve, eliminamos las palabras reservadas function y return y, en algún caso sencillo las llaves {}. En caso de no necesitar parámetros usaremos los paréntesis () vacíos:

```
let error = () => 'Se ha detectado un error';
console.log(error()); //Se ha detectado un error
```

Si necesitamos tan solo 1 parámetro entonces podemos prescindir de los paréntesis

```
var cuadrado = x => x*x;
console.log(cuadrado(2)); // 4
```

NOTA: Las funciones flecha no admiten el uso del objeto arguments sin embargo si que podremos usar el operador de arrastre.

```
let test = () => arguments;
console.log( test( 'foo', 'bar' ) ); // ERROR: arguments is not defined
```

Si hemos de realizar operaciones más complicadas deberemos usar llaves:

```
let foo = (foo, bar, foobar)=>{
let r;
// código de la función
return r;
};
```

PROFUNDIZAR ALGO MÁS CON 7 MANERAS DE CREAR UNA FUNCION:
https://dev.to/victordeandres/funciones-en-javascript-7-formas-de-declarar-una-funcion-523a

4.1.4 Operador Spread

Derivado de la extensión de elementos en otros, viene introducido en la modificación del ECMAScript 2016, por medio de la siguiente sintaxis:

…

Se llama el operador Spread y permite propagar los elementos iterables como son los arrays, strings, objetos, funciones en otro iterable o llamando a una función fecha.

Nos permite trabajar con datos de manera flexible e independientemente del número de parámetros de una función, atributos de un elemento, etc…

Puede utilizarse para concatenar arrays, crear copias superficiales de arrays, convertir cadenas en arrays de caracteres, fusionar o clonar objetos y pasar dinámicamente valores a funciones o constructores, entre otros casos de uso.

Ejemplo para concatenar arrays:

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];

// Concatenar arrays usando operador spread
const concatenatedArr = [...arr1, ...arr2];
console.log(concatenatedArr); // Output: [1, 2, 3, 4, 5, 6]
```

Ejemplo para splitear cadenas:

```
const str = "Hello";
// Operador Spread como una cadena dentro del array
const charArray = [...str];
console.log(charArray); // Output: ['H', 'e', 'l', 'l', 'o']
```

Ejemplo para fusionar o clonar objetos:

```
const obj1 = { a: 1, b: 2 };
const obj2 = { b: 3, c: 4 };

// Fusionar objetos con el operador spread
const mergedObj = { ...obj1, ...obj2 };
console.log(mergedObj); // Output: { a: 1, b: 3, c: 4 }

// Clonar objetos con el operador spread
const clonedObj = { ...obj1 };
console.log(clonedObj); // Output: { a: 1, b: 2 }
```

Ejemplo para emplearlo como argumento en un función flecha:

```
const numbers = [1, 2, 3];
const sum = (a, b, c) => a + b + c;
console.log(sum(...numbers)); // Output: 6
```

El operador spread simplifica las operaciones complejas y permite un código más expresivo y eficiente. Esto lo convierte en una función muy popular entre los desarrolladores de JavaScript.

4.1.5. Ejercicios

Ejercicio:
Defina una función en donde independientemente del número de parámetros recibidos realice las siguientes acciones:

- Muestre con la consola el número total de parámetros recibidos.
- En el caso de recibir más de 2 parámetros, intercambiar los valores del primer y tercer parámetro, mostrando los valores del antes y del después en la consola.

Ejercicio:
Simular un cajero, para ello crea un script en el que tengas una función y empleando parte del diseño de la calculadora:

- Te pide una contraseña o pin para entrar. Botón para validar.
- Botón para: Consultar el saldo disponible, previamente has tenido iniciar con el valor 10.000 €.
- Botón para: sacar dinero, comprobando que no supere el saldo disponible.

4.2. Creación de Objetos y Prototipos

Nos permite modelar elementos abstractos mediante métodos y propiedades.

- Definición de métodos y propiedades.

https://www.w3schools.com/js/js_object_definition.asp

https://www.w3schools.com/js/js_object_properties.asp

https://www.w3schools.com/js/js_object_methods.asp

https://www.w3schools.com/js/js_object_display.asp

https://www.w3schools.com/js/js_object_accessors.asp

https://www.w3schools.com/js/js_object_constructors.asp

https://www.w3schools.com/js/js_object_prototypes.asp

https://www.w3schools.com/js/js_object_es5.asp

Como ya os estáis dando cuento a lo largo de curso, en JS casi todo se puede interpretar como un objeto: números, strings, funciones, fechas, arrays, etc. Ahora en esta sección vamos aprender cómo crear nuestros propios objetos.

Su explicación la trataremos con un ejemplo: Os acabáis de comprar un móvil (nuestro objeto será el móvil). Tiene propiedades como color, peso, tamaño, tamaño de pantalla y marca; también tiene métodos como llamar, tomar una foto, encendido y apagado.

¡Recordatorio! Como ya sabéis:

- Los objetos son una instancia de una clase.
- Las propiedades son una característica, dada en un valor; ya sea tipo String (cadena), numérico, etc.
- Los métodos son una capacidad y para realizarlos utilizamos una función (function)

La programación orientada a objetos es otro de los paradigmas admitidos por JavaScript. A diferencia de otros lenguajes de programación, como Java o C#, JavaScript no utiliza clases, sino que utiliza exclusivamente objetos. Su filosofía es, por tanto, muy distinta de la de esos otros lenguajes.

Una de las características más difíciles de comprender del sistema de objetos de JavaScript es el uso de this. El mecanismo de funcionamiento de this es dinámico, y por ello puede hacer referencia a objetos diferentes en función de cómo se llame a las funciones que lo utilizan. El uso de this no depende, pues, de la definición de la función, sino de cómo se realiza su llamada.

JavaScript también define un mecanismo para enlazar y establecer relaciones entre objetos. Este concepto está relacionado con la herencia en los lenguajes que utilizan clases. Sin embargo, el mecanismo que utiliza JavaScript se basa en la delegación de prototipos (prototypal inheritance). La correcta comprensión de esta característica es imprescindible para poder tener un buen dominio del lenguaje.

Un objeto en JS esta formado por colecciones de datos definidas como propiedades. Es importante conocer que JS tiene objetos pero no tiene clases. Si es posible definir objetos como clases pero no tiene nada que ver con las clases de lenguajes como Java.

En JavaScript encontramos 3 maneras de crear objetos:

4.2.1 Objeto literal declarativo

```
var movil = {Marca:"LG", Peso: 5, pantalla: "5 pulgadas", Color:"Negro"};
console.log(movil)
console.log(movil.Marca)
console.log(movil["Marca"])
```

Es la manera más simple de crear un objeto. Se enmarca dentro de llaves y se describe en pares de (clave: valor).
Para acceder a todas las propiedades del objeto: móvil
Para acceder a cada una de las propiedades: movil.Marca o movil.["Marca"]

4.2.2 Objeto usando la palabra new

```
var movil = new Object();
movil.Marca="LG";
movil.Peso=5;
movil.Pantalla="5 pulgadas";
movil.Color="Negro";
```

El operador new crea una instancia de un tipo de objeto a partir de una función constructora. Recordemos que en JavaScript un constructor es una función.

4.2.3 Objeto haciendo una función constructora

```
 <p id="elementoprueba">
 <script type="text/javascript">

function movil(marca, peso, pantalla, color) {
  this.Marca = marca;
  this.Peso = peso;
  this.Pantalla = pantalla;
  this.Color = color;
}
var compra = new movil("LG", 5, "5 pulgadas", "Negro");
document.getElementById("elementoprueba").innerHTML = compra.Marca + " ---
 " + compra["Marca"];
</script>
```

Una función constructora crea una copia de un objeto específico; en otros lenguajes de programación son conocidas como clases.

Ahora vamos a agregar métodos. Existen diversas maneras de realizar este proceso, pero tomaremos de ejemplo la manera más simple: utilizando la palabra this. Se puede utilizar en un método para referirnos al objeto actual.

```
<p id="elementoprueba">
<script type="text/javascript">
let movil = { Marca: "LG", encendido : function(c) {
return "Mi movil marca" +" "+ this.Marca +" "+ "esta encendido";
}};
document.getElementById("elementoprueba").innerHTML = movil.encendido();
</script>
```

4.2.4 Ejercicios

1) Objetos construidos
Imaginemos que vamos a desarrollar un juego y debemos crear un objeto "Jugador" que tenga una propiedad "fuerza" que inicialmente tenga el valor 1, un método "incrementarFuerza" que nos permita incrementar en 1 la fuerza del jugador y un método "consultarFuerza" que nos muestre un mensaje con la fuerza del jugador.

```
function Jugador() {
this.fuerza = 1;
this.mostrarFuerza = function () {
console.log("Tu fuerza es de " + this.fuerza);
};
this.incrementarFuerza = function () {
this.fuerza += 1;
};
}

let jugador1 = new Jugador();
jugador1.mostrarFuerza(); // Tu fuerza es de 1
jugador1.incrementarFuerza();
jugador1.mostrarFuerza(); // Tu fuerza es de 2
```

2) Objetos declarativos
Idem del ejercicio 1 pero con objetos declarativos.

```
let jugador = {
fuerza: 1,
incrementarFuerza: function () {
this.fuerza = this.fuerza + 1;
},
consultarFuerza: function () {
console.log(this.fuerza);
},
};
jugador.consultarFuerza();
jugador.incrementarFuerza();
jugador.consultarFuerza();

let movil = { Marca: "LG", encendido : function(c) {
return "Mi movil marca" +" "+ this.Marca +" "+ "esta encendido";
}};
document.getElementById("elementoprueba").innerHTML = movil.encendido();
```

3) Objetos con la palabra reservada Object
Idem del ejercicio 1 pero empleando la palabra reservada Object.

```
var jugador = new Object({
fuerza: 1,
incrementarFuerza: function (fuerza) {
this.fuerza = this.fuerza + 1;
},
consultarFuerza: function (fuerza) {
console.log(this.fuerza);
},
});
jugador.consultarFuerza();
jugador.incrementarFuerza();
jugador.consultarFuerza();
```

4) Crear media de objetos
Supuesto: Primer día de Prácticas en FCTs:
Se os pide crear una función constructora para crear objetos de una clase especifica.
Para ello tendrás que crearte un Objeto Persona que tendrá una serie de características: nombre, apellido, edad, email.
Crea 4 objetos de esta misma clase con los siguientes atributos:
Roberto Martínez: Tiene 22 años y su contacto es roberto.martinez@tuempresa.com
Antonio López: Tiene 25 años y su contacto es antonio.lopez@tuempresa.com
Javier Rodríguez: Tiene 18 años y su contacto es javier.rodriguez@tuempresa.com
Eva Teruel: Tiene 33 años y su contacto es eva.teruel@tuempresa.com
Calcula la edad media de las 4 personas.

4.2.5 Módulos

Cuando la complejidad de las aplicaciones aumenta, se hace necesario disponer de un mecanismo que nos permita dividir el código en diferentes archivos. Las ventajas de ello son variadas: mejor organización, agrupación de código relacionado con la misma funcionalidad, disminución de conflictos de versiones al poder trabajar diferentes personas en distintos archivos, etcétera.

Es muy probable también que sea necesario utilizar librerías externas. De nuevo, es deseable disponer de los medios para cargar dichas librerías de una forma sencilla, de tal manera que su código interno permanezca convenientemente aislado para evitar conflictos en nombres de variables o funciones, de forma que sea accesible únicamente la interfaz o API de dicha librería.

Este mecanismo recibe el nombre de sistema de módulos. Durante muchos años, JavaScript no ha tenido ningún sistema de módulos estándar en el navegador, sino que ha utilizado librerías externas creadas para ello o ha adoptado el sistema creado para NodeJS. Finalmente, el estándar EcmaScript ha desarrollado su propio sistema, que ha sido incluido en JavaScript y en la actualidad es compatible con la mayoría de los navegadores, así como NodeJS.

Ejercicio con Módulos:
Crea un módulo denominado en un archivo con extensión .js y almacena el código JavaScript creado. Deja únicamente las definiciones de los prototipos y expórtalas para que puedan ser utilizadas mediante import. A continuación, crea una página HTML que contenga un script que importe dicho módulo y cree un enemigo de cada tipo utilizando las funciones importadas. Ver ejercicio “4.2.5 Ejercicio Modulos”.

4.3 Programación Funcional

Como hemos visto en la Unidad Didáctica 3, la programación con bucles tiene grandes acepciones a la hora de recorrer matrices, tales como condiciones del tipo if, else ó else if; y recorridos del tipo for, for-in, for-of, do-while, while o switch entre otros.

Ahora bien, la inclusión de la gran cantidad de bucles puede generar código complicado de seguir, mantener y mas proclive a equivocarse. JavaScript es compatible con el paradigma de programación funcional. Este estilo de programación tiene como característica principal el uso de funciones que devuelven datos que, a su vez, pueden ser utilizados por otras funciones. Las funciones suelen estar altamente especializadas y, en la medida de lo posible, minimizan los efectos secundarios, es decir, no provocan cambios en variables externas ni en los parámetros que se les pasan, sino que se limitan a devolver un resultado nuevo.

En la programación funcional, las funciones se consideran ciudadanas de primera clase (first-class citizens): reciben el mismo tratamiento que cualquier otro tipo de datos, por lo que se pueden pasar como parámetros a otras funciones y se pueden almacenar en variables.

En la programación funcional es habitual utilizar funciones para realizar tareas que en la programación procedimental se realizarían con bucles. Los usos más habituales están relacionados con el tratamiento de los datos almacenados en arrays, como la búsqueda, filtrado, transformación o cálculo de datos agregados.

Por ello, y para evitar el uso masivo de los bucles for detalladas anteriormente, en la versión de ECMAScript 2016 tenemos los siguientes **ITERADORES**:

El Array de ejemplo que usamos:

```
const reservas = [
{ id: 10000, precio: 25, habitacion: 'individual', pagada: false },
{ id: 10001, precio: 15, habitacion: 'individual', pagada: false },
{ id: 10002, precio: 55, habitacion: 'doble', pagada: true },
{ id: 10003, precio: 55, habitacion: 'doble', pagada: true },
{ id: 10004, precio: 65, habitacion: 'doble', pagada: true } ];
```

4.3.1 Every

Devuelve true si todos los elementos del array cumplen con una condición dada. Este método te ahorra implementar bucles for con un if - break.

Ejemplo: comprobar si todas las reservas están pagadas.

```
const reservaPagadas = reservas.every(booking => booking.pagada);
console.log(reservaPagadas);
```

4.3.2 Filter

Extrae de un Array aquellos elementos que cumplan cierta condición.

Ejemplo: reservas que no están pagadas.

```
const nopagadas = reservas.filter(booking => !booking.pagada);
console.log(nopagadas);
```

Ejemplo: reservas por encima de los 50 euros.

```
const reservaCara = reservas.filter(booking => booking.precio > 50);
console.log(reservaCara);
```

4.3.3 Find

Devuelve el primer elemento que cumpla con una condición dada. Y sino hay ninguna que lo cumpla, devolverá undefined.

Ejemplo:

```
const booking = reservas.find(booking => booking.habitacion === 'individual');
console.log(booking.id);
```

4.3.4 FindIndex

Similar a Find, devuelve el índice del Array del primer elemento que cumple con la condición. Y sino no hay ningún elemento devolverá -1.

Ejemplo:

```
const reservaIndex = reservas.findIndex(booking => booking.habitacion === 'doble');
console.log(reservas[reservaIndex].id);
```

4.3.5 Map

Aplica a cada elemento del array una función que modifica cada elemento del array.

Ejemplo: un cliente introduce un cupón de descuento y se debe aplicar un descuento del 10% en cada reserva:

```
const descuento = 10;
const descuentoReserva = reservas.map(booking => ({
price: booking.precio * (1 - (descuento / 100))
}));
console.log(descuentoReserva);
```

4.3.6 Reduce

Similar a Map, pero en vez de devolver un array, lo que devuelve es un total, es decir aplica una función y va acumulando el resultado.

Ejemplo: Sumar el total de euros de las reservas que hay en el array:

```
const totalPrecio = reservas.reduce((total, booking) => total + booking.precio, 0);
console.log(totalPrecio);
```

4.3.7 Some

Similar a Every, pero en este caso te devuelve true si al menos uno de los elemento cumple con la condición que le pasamos.

Ejemplo: Buscar si el cliente tiene al menos una habitacion 'suit' realizada:

```
const habitacionSuit = reservas.some(booking => booking.habitacion === 'suit');
console.log(habitacionSuit);
```

4.3.8 Set

¿Cuántas veces hemos querido crear un array con datos únicos?. Una solución puede ser generar un nuevo array iterando elemento a elemento comprobando que no estuviese ya metido. Una de las principales características de Set es que un valor sólo puede estar una sola vez, por lo que resulta bastante conveniente para eliminar valores duplicados.

Ejemplo: Dado el Array de reservas, añadir una nueva reserva más de una vez.

```
const reservaNueva = {
id:10006,
precio: 350,
habitacion: 'individual',
pagado: false,
};
const reservaUnica = new Set(reservas);
reservaUnica.add(reservaNueva);
reservaUnica.add(reservaNueva);
reservaUnica.add(reservaNueva);
reservaUnica.add(reservaNueva);
console.log(reservaUnica);
```

4.3.9 Delete

Con el método Add se añade nuevos elementos a nuestro Set. Con el método Delete se elimina un valor ya introducido. Si se ha eliminado del Set devolverá true, si no existía devolverá false.

Ejemplo: Eliminar la reserva anterior:

```
const reservaNueva = {
id:10006,
precio: 350,
habitacion: 'individual',
pagado: false,
};

const reservaUnica = new Set(reservas);
reservaUnica.add(reservaNueva);
console.log('Nuevo booking eliminado', reservaUnica.delete(reservaNueva));
console.log('Eliminar nuevo booking de nuevo', reservaUnica.delete(reservaNueva));
console.log(reservaUnica);
```

4.3.10 Size

La propiedad Size es la equivalente a Length de los arrays para comprobar el tamaño del Set.

Ejemplo: Mostrar el total de las reservas de nuestro Set de reservas:

```
const reservaUnica = new Set(reservas);
console.log(reservaUnica.size);
```

4.3.11 Has

Para comprobar si Set tiene ya un valor el método Has es el apropiado. Este método nos devuelve true o false dependiendo de si el elemento ya está o no.

Ejemplo: Añadir una nueva reserva y comprobar si se ha añadido correctamente:

```
const reservaNueva = {
id:10006,
precio: 350,
habitacion: 'individual',
pagado: false,
};
const reservaUnica = new Set(reservas);
reservaUnica.add(reservaNueva);
console.log(reservaUnica.has(reservaNueva));
```

4.3.12 Clear

Si queremos vaciar el Set sin tener que crear otro nuevo podemos hacerlo usando el método Clear.

Ejemplo: Vaciar el set que tenemos de reservas:

```
const reservaUnica = new Set(reservas);
reservaUnica.clear();
console.log(reservaUnica);
```

4.4 Ejercicios

1) Crear un objeto "Cuenta Bancaria" con las propiedades "titular", "saldo" y "intereses" y los métodos "depositar", "retirar" y "calcularIntereses". Calcular interés, deposito y retirada de dinero.

2) Crear un objeto "Libro" con las propiedades "titulo", "autor", "cantidadPaginas" y el método "existePagina". Si existe la Página, mostrar todas las propiedades del objeto.

3) Crear un objeto "Vehiculo" con las propiedades "marca", "modelo", "anio" y el método "obtenerEdad"

4) Crear un objeto "Receta" con las propiedades "nombre", "ingredientes" y el método "imprimirIngredientes"

5) Crear un objeto "Persona" con las propiedades "nombre", "edad" y el método "esMayorDeEdad"

6) Crear un objeto calculadora con los métodos sumar, restar, dividir y multiplicar que estén definidos con funciones flecha.

7) Crear un objeto “gestor de tareas” que tenga una propiedad tareas y unos métodos para agregar tareas, eliminar tareas y mostrar tareas. Todos ellos definidos con funciones flecha.

8) Crear un gestor de cuentas que tenga una clase heredada del objeto principal que tenga dos propiedades: nombre y saldo. Además se podrá agregar cuenta, obtener el saldo, depositar saldo y retirar saldo. Todo ello con funciones flecha.

9) Empleando iteradores complejos, obtener el promedio de una lista de números.

```
const numeros = [10, 20, 30, 40];
const suma = numeros.reduce((a, b) => a + b);
const promedio = suma / numeros.length;
console.log(promedio); // Output: 25
```

10) Obtener una lista de palabras únicas de una frase

```
const frase = "Esta es una frase de prueba para el ejemplo de iteradores en JavaScript";
const palabrasUnicas = [...new Set(frase.split(" "))];
console.log(palabrasUnicas); // Output: ["Esta", "es", "una", "frase", "de", "prueba", "para", "el", "ejemplo", "en",
"JavaScript"]
```

11) Obtener el máximo valor de una lista de números

```
const numeros = [10, 5, 20, 15, 8];
const maximo = Math.max(...numeros);
console.log(maximo); // Output: 20
```

12) Obtener la lista de empleados con un salario mayor a $5000:

```
const empleados = [
  { nombre: "Juan", salario: 4500 },
  { nombre: "María", salario: 6000 },
  { nombre: "Pedro", salario: 3500 },
  { nombre: "Luis", salario: 7000 }
];
const empleadosSalarioMayor5000 = empleados.filter(empleado => empleado.salario > 5000);
console.log(empleadosSalarioMayor5000); // Output: [{ nombre: "Maria", salario: 6000 }, { nombre: "Luis", salario: 7000 }]
```