¿Qué son los ARRAYS en programación?

Los arrays son datos en memoria donde 1 sola variable puede almacenar múltiples valores

- También se les llama vectores
- Pueden agregárseles elementos, removerlos o accederlos de manera aleatoria (cualquier posición en cualquier momento)

Con los arrays se logra:

- Mantener en memoria la información de muchos objetos; hasta ahora sólo se mantenía en memoria el objeto que estaba siendo procesado
- Realizar consultas que de otra manera es imposible ya que amerita un doble recorrido
- Ej.: para saber cuáles estudiantes de una sección tienen edad por encima del promedio es preciso:
  - ✓ PRIMERO: calcular el promedio (esto amerita un ciclo completo a todos los datos)
  - ✓ SEGUNDO: ver en cada estudiante su edad y compararla con el promedio, contarlo si la edad es mayor que el promedio
- Otros ejemplos:
  - ✓ Cuántos estudiantes tienen la edad mayor
  - ✓ Cuántos estudiantes tienen la misma edad que el último estudiante

Las estructuras de datos que hemos usado hasta ahora

Así hemos trabajado hasta ahora: leemos un objeto y de inmediato procesamos los datos de procesos universales.

estud
nombre: Luis
edad: 18

estud
nombre: Ana
edad: 20

estud
nombre: Eva
edad: 16

estud
nombre: Ray
edad: 20

- El mismo único objeto
- Al procesar un objeto, se pierden los datos del anterior
- ¿Quienes están por encima del promedio?
- Para saberlo, debo leer todos los objetos.
- Luego volver a ver cada objeto y compararlo con la edadPromedio.
- IMPOSIBLE ya que he perdido los datos anteriores.

edadPromedio
= (18 + 20 + 16 + 20) / 4 = 18.5

acumEdad cntEstuds

Con los objetos Cl_estud y Cl_salon vamos procesando uno a uno y calculamos los atributos necesarios para determinar el requerimiento, que en este caso es el proceso universal de promedio (acumulador / contador). Para el ejemplo: acumEdad y cntEstuds

```
class Cl_salon{
  constructor(){
     this.acumEdad
     = this.cntEstuds = 0
  }
  procesarEstud(estud){
     this.cntEstuds++
     this.acumEdad += estud.edad
  }
  edadPromedio(){
    return this.acumEdad / this.cntEstuds
  }
}
```

¡ ¡ ACÁ ! !

Procesas el estud, y luego se pierden sus datos

- ¿Cuántos están encima del promedio?... El promedio es 18.5, así que 2 están por encima. ¿Quiénes por encima? -> Ana y Ray.
- ¿Quién o quiénes tienen la mayor edad?... La mayor edad es 20, y la tienen Ana y Ray
- ¿Los nombre de quienes tengan una edad dada por el usuario?... Digamos 20 años: [Ana, Luis]
- **PARA ESTOS CÁLCULOS, PRIMERO DEBE CARGARSE TODA LA INFO**

Operaciones con los arrays

Operaciones posibles con arrays de datos en JavaScript

Creación de Arrays

En JavaScript, puedes crear arrays de varias maneras:

- Usando corchetes ([]):

```
let arrayVacio = [];
let arrayConElementos = [1, 2, 3, 4, 5];
```

- Usando el constructor Array:

```
let arrayVacio = new Array();
let arrayConElementos = new Array(1, 2, 3, 4, 5);
```

Asignación de Elementos

Puedes asignar elementos a un array en el momento de su creación o después:

- En el momento de la creación:

```
let frutas = ["manzana", "banana", "cereza"];
```

- Después de la creación:

```
let numeros = [];
numeros[0] = 1;
numeros[1] = 2;
numeros[2] = 3;
```

Acceso a Elementos

Para acceder a los elementos de un array, utilizas el índice del elemento entre corchetes []. Los índices comienzan en 0.

- Acceder a un elemento específico:

```
let frutas = ["manzana", "banana", "cereza"];
console.log(frutas[0]); // "manzana"
console.log(frutas[1]); // "banana"
console.log(frutas[2]); // "cereza"
```

- Modificar un elemento:

```
frutas[1] = "naranja";
console.log(frutas); // ["manzana", "naranja", "cereza"]
```

Programa demo: https://2v2jrq.csb.app

Repositorio demo: https://github.com/g-torrealba-ucla/L25.1.p.c2.12-Demo-operaciones-Arrays

Operaciones con los arrays

Operaciones posibles con arrays de datos en JavaScript

Ejemplos con Tipos de Datos Primitivos y Objetos

- Tipos de Datos Primitivos

```
let numeros = [1, 2, 3, 4, 5];
let booleanos = [true, false, true];
let strings = ["hola", "mundo"];


console.log(numeros[2]); // 3
console.log(booleanos[1]); // false
console.log(strings[0]); // "hola"
```

- Objetos:

```
let personas = [
    { nombre: "Leo", edad: 19 },
    { nombre: "Ana", edad: 20 },
    { nombre: "Eva", edad: 17 }
];
console.log(personas[0].nombre); // "Leo"
console.log(personas[1].edad); // 20

// Modificar un objeto en el array
personas[2].edad = 18;
console.log(personas[2].edad); // 18
```

Ver demo: A. Ejemplo creación de arrays

Resumen de Operaciones Básicas

- Creación: Usando [] o new Array().

- Asignación: Directamente al crear el array o después usando índices.

- Acceso: Utilizando índices entre corchetes: [].

Resumen de Operaciones Básicas

Además de la creación, asignación y acceso, podrías considerar:

- Iteración: Recorrer los elementos del array usando bucles (for, forEach, map, etc.).

- Métodos de Array: Métodos como push, pop, shift, unshift, splice, slice, concat, filter, map, reduce, etc., para manipular arrays de diversas maneras.

Programa demo: https://2v2jrq.csb.app

Repositorio demo: https://github.com/g-torrealba-ucla/L25.1.p.c2.12-Demo-operaciones-Arrays

Operaciones con los arrays

Operaciones posibles con arrays de datos en JavaScript

Reglas Básicas para Acceder a Elementos en un Array de Objetos

Acceder a los elementos de un array de objetos en JavaScript es bastante sencillo una vez que entiendes las reglas básicas. Aquí te explico cómo hacerlo, con ejemplos claros:

- Acceso por Índice:

- Los arrays en JavaScript son indexados, lo que significa que cada elemento tiene una posición numérica, comenzando desde 0.
- Para acceder a un elemento específico, usas el índice del array entre corchetes [].

- Acceso a Propiedades del Objeto:

- Una vez que has accedido al objeto dentro del array, puedes acceder a sus propiedades utilizando la notación de punto (.) o la notación de corchetes [].

Creación de un Array de Objetos

```
let personas = [
    { nombre: "Leo", edad: 19 },
    { nombre: "Ana", edad: 20 },
    { nombre: "Eva", edad: 17 }
];
```

Acceso a Elementos del Array

Acceder al Primer Objeto:

```
let primeraPersona = personas[0];
console.log(primeraPersona); // { nombre: "Leo", edad: 19 }
```

Acceder a una Propiedad de un Objeto:

```
let nombrePrimeraPersona = personas[0].nombre;
console.log(nombrePrimeraPersona); // "Leo"
```

Acceder a una Propiedad Usando Notación de Corchetes:

```
let edadPrimeraPersona = personas[0]["edad"];
console.log(edadPrimeraPersona); // 19
```

Ver demo: B. Ejemplo acceso de elementos

Ejemplo Completo con Tipos de Datos Primitivos y Objetos

- Array de Tipos Primitivos

```
let numeros = [1, 2, 3, 4, 5];
console.log(numeros[2]); // 3
```

- Array de Objetos

```
let desarrolladores = [
  { nombre: "John", edad: 25, lenguajes: ["JavaScript", "Python"] },
  { nombre: "Kelly", edad: 37, lenguajes: ["Ruby", "Python", "C", "C++"] },
  { nombre: "Zack", edad: 45, lenguajes: ["Go", "C#"] }
];

Acceder al nombre del primer desarrollador
console.log(desarrolladores[0].nombre); // "John"

Acceder al segundo lenguaje del segundo desarrollador
console.log(desarrolladores[1].lenguajes[1]); // "Python"
```

Ver demo: C. Ejemplo acceso de elementos

Resumen

- **Creación**: Puedes crear arrays de objetos usando corchetes ([]) y definiendo objetos dentro.
- **Asignación**: Los elementos se asignan directamente al crear el array o después usando índices.
- **Acceso**: Utiliza índices para acceder a los objetos y notación de punto o corchetes para acceder a las propiedades de los objetos.

Operaciones con los arrays

Operaciones posibles con arrays de datos en JavaScript

Operaciones de Acceso y Modificación

Agregar Elementos:

- **push**(): Añade uno o más elementos al final del array.
- **unshift**(): Añade uno o más elementos al principio del array.

Eliminar Elementos:

- **pop**(): Elimina el último elemento del array.
- **shift**(): Elimina el primer elemento del array.
- **splice**(): Elimina elementos en una posición específica.

Modificar Elementos:

- **Acceder y modificar**: Utilizando el índice del array, por ejemplo, array[0] = nuevoValor.

Operaciones de Filtrado y Agrupación

Filtrar Datos:

- **filter**(): Crea un nuevo array con todos los elementos que cumplan una condición específica.

Agrupar Datos:

- **reduce**(): Puede ser utilizado para agrupar elementos en un array basado en una condición.

Operaciones de Cálculo

Calcular Sumas/Promedios:

- **reduce**(): Suma todos los elementos del array o calcula el promedio.
- **map**(): Aplica una función a cada elemento del array y retorna un nuevo array con los resultados.

Contar Elementos:

- **length**: Devuelve el número de elementos en el array.

Operaciones de Ordenación

Ordenar Elementos:

- **sort()**: Ordena los elementos del array en su lugar y retorna el array.
- **reverse()**: Invierte el orden de los elementos del array.

Operaciones con los arrays

Ejemplos con push y unshift

El método push añade uno o más elementos al final de un array.

- **Antes de push**

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

- **Después de push**

```
numeros.push(4);
console.log(numeros); // [1, 2, 3, 4]

numeros.push(5, 6);
console.log(numeros); // [1, 2, 3, 4, 5, 6]
```

Ver demo: D. Ejemplo push

El método unshift añade uno o más elementos al principio de un array.

- **Antes de unshift**

```
let frutas = ["manzana", "banana"];
console.log(frutas); // ["manzana", "banana"]
```

- **Después de unshift**

```
frutas.unshift("naranja");
console.log(frutas); // ["naranja", "manzana", "banana"]

frutas.unshift("uva", "pera");
console.log(frutas); // ["uva", "pera", "naranja", "manzana",
"banana"]
```

Ver demo: E. Ejemplo unshift

- **Resumen**

push: Añade elementos al final del array.

unshift: Añade elementos al principio del array.

Estos métodos son muy útiles para manipular arrays de manera dinámica.

Ejemplos con pop

El método pop elimina el último elemento del array.

- **Ejemplo de uso de pop**

```
// Array de elementos primitivos (números)
let numeros = [1, 2, 3, 4, 5];
let ultimoNumero = numeros.pop();

console.log(numeros); // [1, 2, 3, 4]
console.log(ultimoNumero); // 5
```

En este ejemplo, el método pop() elimina el último elemento del array numeros y lo devuelve. El array original se modifica y el valor eliminado se almacena en ultimoNumero.

```
// Array de objetos
let personas = [
  { nombre: "Ana", edad: 25 },
  { nombre: "Luis", edad: 30 },
  { nombre: "Carlos", edad: 35 }
];
let ultimaPersona = personas.pop();

console.log(personas);
// [{ nombre: "Ana", edad: 25 }, { nombre: "Luis", edad: 30 }]
console.log(ultimaPersona); // { nombre: "Carlos", edad: 35 }
```

Ver demo: F. Ejemplo pop

En este caso, pop() elimina el último objeto del array personas y lo devuelve. El array original se modifica y el objeto eliminado se almacena en ultimaPersona.

El método pop() es muy útil cuando necesitas trabajar con el último elemento de un array, ya sea para eliminarlo o para realizar alguna operación con él

Operaciones con los arrays

Ejemplos con shift

El método shift elimina el primer elemento del array.

- **Ejemplo de uso de shift**

```
// Array de elementos primitivos (números)
let numeros = [1, 2, 3, 4, 5];
let primerNumero = numeros.shift();

console.log(numeros); // [2, 3, 4, 5]
console.log(primerNumero); // 1
```

En este ejemplo, el método shift() elimina el primer elemento del array numeros y lo devuelve. El array original se modifica y el valor eliminado se almacena en primerNumero.

```
// Array de objetos
let personas = [
  { nombre: "Ana", edad: 25 },
  { nombre: "Luis", edad: 30 },
  { nombre: "Carlos", edad: 35 }
];
let primeraPersona = personas.shift();

console.log(personas);
// [{ nombre: "Luis", edad: 30 }, { nombre: "Carlos", edad: 35 }]
console.log(primeraPersona); // { nombre: "Ana", edad: 25 }
```

Ver demo: G. Ejemplo shift

En este caso, shift() elimina el primer objeto del array personas y lo devuelve. El array original se modifica y el objeto eliminado se almacena en primeraPersona.

El método shift() es útil cuando necesitas trabajar con el primer elemento de un array, ya sea para eliminarlo o para realizar alguna operación con él

Ejemplos con splice

El método splice elimina elementos en una posición específica del array.

- **Ejemplo de uso de splice**

```
// Array de elementos primitivos (números)
let numeros = [1, 2, 3, 4, 5];
let eliminados = numeros.splice(2, 2, 6, 7);

console.log(numeros); // [1, 2, 6, 7, 5]
console.log(eliminados); // [3, 4]
```

En este ejemplo, splice() elimina dos elementos a partir del índice 2 y agrega los números 6 y 7 en su lugar. El array original se modifica y los elementos eliminados se almacenan en eliminados.

```
// Array de objetos
let personas = [
  { nombre: "Ana", edad: 25 },
  { nombre: "Luis", edad: 30 },
  { nombre: "Carlos", edad: 35 },
  { nombre: "María", edad: 28 }
];
let eliminados = personas.splice(1, 2, { nombre: "Pedro", edad: 40 });

console.log(personas);
// [{ nombre: "Ana", edad: 25 }, { nombre: "Pedro", edad: 40 },
{ nombre: "María", edad: 28 }]
console.log(eliminados);
// [{ nombre: "Luis", edad: 30 }, { nombre: "Carlos", edad: 35 }]
```

Ver demo: H. Ejemplo splice

En este caso, splice() elimina dos objetos a partir del índice 1 y agrega un nuevo objeto en su lugar. El array original se modifica y los objetos eliminados se almacenan en eliminados.

El método splice() es muy versátil y permite tanto eliminar como agregar elementos en cualquier posición del array

La función flecha en JavaScript

Las funciones flecha en JavaScript son una forma más concisa de escribir funciones. Tienen una sintaxis más compacta que las funciones tradicionales.

- **Sintaxis Básica**

La sintaxis de una función flecha es:

```
(param1, param2, ..., paramN) => expresión
```

Si la función solo tiene un parámetro, puedes omitir los paréntesis:

```
param => expresión
```

Si no tiene parámetros, se usan paréntesis vacíos:

```
() => expresión
```

- **Ejemplos: Función Tradicional vs Función Flecha**

```
// Función Tradicional:
function sumar(a, b) {
  return a + b;
}
console.log(sumar(2, 3)); // 5

// Función Flecha:
const sumar = (a, b) => a + b;
console.log(sumar(2, 3)); // 5

// Sin Parámetros
const saludar = () => console.log("¡Hola!");
saludar(); // ¡Hola!
```

```
// Un Solo Parámetro
const cuadrado = x => x * x;
console.log(cuadrado(4)); // 16

// Con Cuerpo de Bloque
// Si necesitas más de una línea de código, usa llaves y return:
const sumarYMultiplicar = (a, b) => {
  const suma = a + b;
  return suma * 2;
};
console.log(sumarYMultiplicar(2, 3)); // 10
```

- **Características Especiales**

No tiene su propio this: Las funciones flecha no tienen su propio contexto this, sino que heredan el this del contexto en el que fueron definidas.

No se puede usar como constructor: No puedes usar funciones flecha con new.

Las funciones flecha son muy útiles para escribir funciones cortas y mantener el código limpio y legible

La función flecha... Con Arrays

Las funciones flecha y los métodos de arrays son una combinación poderosa en JavaScript, para transformar arrays de manera concisa y eficiente.

- **Método map()**

El método map() crea un nuevo array con los resultados de aplicar una función a cada uno de los elementos del array original. Aquí tienes un ejemplo básico:

```
const numeros = [1, 2, 3, 4, 5];
const cuadrados = numeros.map(num => num * num);
console.log(cuadrados); // [1, 4, 9, 16, 25]
```

- **Ejemplo con Elementos Primitivos**

Supongamos que tienes un array de números y quieres duplicar cada número:

```
const numeros = [1, 2, 3, 4, 5];
const duplicados = numeros.map(num => num * 2);
console.log(duplicados); // [2, 4, 6, 8, 10]
```

- **Ejemplo con Objetos**

Ahora, imagina que tienes un array de objetos y quieres extraer una propiedad específica de cada objeto:

```
const personas = [
  { nombre: "Ana", edad: 25 },
  { nombre: "Luis", edad: 30 },
  { nombre: "Carlos", edad: 35 }
];
const nombres = personas.map(persona => persona.nombre);
console.log(nombres); // ["Ana", "Luis", "Carlos"]
```

Ver demo: I. Método map() -1-

- **Ejemplo Complejo**

Supongamos que quieres incrementar la edad de cada persona en 1 año:

```
const personas = [
  { nombre: "Ana", edad: 25 },
  { nombre: "Luis", edad: 30 },
  { nombre: "Carlos", edad: 35 }
];
const personasConEdadIncrementada = personas.map(persona => ({
  ...persona,
  edad: persona.edad + 1
}));
console.log(personasConEdadIncrementada);
// [{ nombre: "Ana", edad: 26 },
    { nombre: "Luis", edad: 31 },
    { nombre: "Carlos", edad: 36 }]
```

Ver demo: J. Método map() -2-

En este último ejemplo, usamos el operador de propagación (...) para copiar las propiedades del objeto original y luego modificamos la propiedad edad.

Las funciones flecha y el método map() son herramientas muy útiles para trabajar con arrays de manera eficiente y clara

Otras operaciones y métodos... Con Arrays

Recorridos con ciclos tradicionales

```
// Con ciclo for
// Definimos un array de números
let numeros = [1, 2, 3, 4, 5];

// Usamos un bucle for para recorrer el array
for (let i = 0; i < numeros.length; i++) {
    console.log(numeros[i]);
}

// Con ciclo while
let frutas = ['manzana', 'banana', 'cereza', 'durazno'];
let i = 0;

while (i < frutas.length) {
  console.log(frutas[i]);
  i++;
}

// Con ciclo do-while
let arr = [1, 2, 3, 4, 5];
let i = 0;

do {
    console.log(arr[i]);
    i++;
} while (i < arr.length);
```

Ver demo: K. Ejemplo con ciclos tradicionales

Método forEach()

El método forEach en JavaScript se utiliza para ejecutar una función específica una vez por cada elemento de un array. Aquí tienes un resumen de cómo funciona:

```
// Sintaxis
array.forEach(function callback(currentValue, index, array) {
  // tu iterador
}
```

Parámetros

- **callback**: La función a ejecutar en cada elemento, que recibe tres argumentos:
- **currentValue**: El valor del elemento actual.
- **index (opcional)**: El índice del elemento actual.
- **array (opcional)**: El array sobre el que se está aplicando forEach.

- **Ejemplo**

```
// Mock Data
const mockData = [10, 20, 30, 40, 50];

// Generated Code
mockData.forEach(function(item) {
  console.log(item);
});

// Using arrow function
mockData.forEach(item => console.log(item));
```

Ver demo: L. Ejemplo con método forEach

Programa demo: https://2v2jrq.csb.app

Repositorio demo: https://github.com/g-torrealba-ucla/L25.1.p.c2.12-Demo-operaciones-Arrays

Otros métodos... Con Arrays

Método filter()

- **Ejemplo con Elementos Primitivos**

Supongamos que tienes un array de números y quieres obtener solo los números mayores a 10:

```
const numeros = [5, 12, 8, 130, 44];
const mayoresQueDiez = numeros.filter(num => num > 10);
console.log(mayoresQueDiez); // [12, 130, 44]
```

En este ejemplo, filter() crea un nuevo array que contiene solo los números del array original que son mayores a 10.

- **Ejemplo con Objetos**

Ahora, imagina que tienes un array de objetos y quieres filtrar solo aquellos objetos que cumplen una cierta condición. Por ejemplo, obtener solo las personas mayores de 18 años:

```
const personas = [
  { nombre: "Ana", edad: 25 },
  { nombre: "Luis", edad: 15 },
  { nombre: "Carlos", edad: 35 }
];
const mayoresDe18 = personas.filter((persona) => persona.edad > 18);
console.log(mayoresDe18);
// [{ nombre: "Ana", edad: 25 },
   { nombre: "Carlos", edad: 35 }]
```

Ver demo: M. Ejemplo con método filter

En este caso, filter() crea un nuevo array que contiene solo los objetos que representan a personas mayores de 18 años.

Método sort()

- **Ejemplo con Elementos Primitivos**

```
let numeros = [10, 5, 8, 1, 7];
numeros.sort((a, b) => a - b); // Ordena de menor a mayor
console.log(numeros); // [1, 5, 7, 8, 10]

let palabras = ["manzana", "naranja", "banana", "kiwi"];
palabras.sort(); // Ordena alfabéticamente
console.log(palabras); // ["banana", "kiwi", "manzana", "naranja"]
```

En este ejemplo, filter() crea un nuevo array que contiene solo los números del array original que son mayores a 10.

- **Ejemplo con Objetos**

Para ordenar un array de objetos, necesitas proporcionar una función de comparación que defina cómo se deben ordenar los objetos. Por ejemplo, si tienes un array de objetos que representan personas y quieres ordenarlos por edad:

```
let personas = [
    { nombre: "Ana", edad: 25 },
    { nombre: "Luis", edad: 30 },
    { nombre: "Carlos", edad: 20 }
];

personas.sort((a, b) => a.edad - b.edad); // Ordena por edad de
menor a mayor
console.log(personas);
// [
//     { nombre: "Carlos", edad: 20 },
//     { nombre: "Ana", edad: 25 },
//     { nombre: "Luis", edad: 30 }
// ]
```

Ver demo: N. Ejemplo con método sort

La estructura de datos ARRAY

Así hemos trabajado hasta ahora: leemos un objeto y de inmediato procesamos los datos de procesos universales.

Ahora el objeto Cl_salon contiene un array con Cl_estud, y lo vamos cargando dinámicamente (agregar / eliminar). Al requerir un resultado, **se recorre el arreglo para calcularlo**.

estuds

nombre: Luis edad: 18
nombre: Ana edad: 20
nombre: Eva edad: 16
nombre: Ray edad: 20

- Ahora los objetos se guardan TODOS dentro del ARRAY
- El ARRAY tiene varias posiciones, comenzando en 0
- Cada posición se accede con corchetes. Ej. estuds[0]
- Al referirnos a **estuds**, es todo el arreglo.
- Al referirnos a una posición, ej. **Estuds[1]**, es un objeto
- Al referirnos a un componente, Ej. **Estuds[2]**.edad accedemos al atributo o método del objeto

edadPromedio = SE RECORRE EL ARRAY

- ✓ En un método de la clase Cl_salon
- ✓ acumEdad y cntEstuds serán variables locales al método

```
class Cl_salon{
  constructor(){
    this.estuds = []
  }
  agregarEstud(estud){
    this.estuds.push(estud)
  }
  eliminarEstud(index){
    this.estuds.splice(index,1)
  }
  edadPromedio(){
    let acumEdad = 0, cntEstuds = 0
    this.estuds.forEach((estud)=>{
      acumEdad += estud.edad
      cntEstuds++
    })
    return acumEdad / cntEstuds
  }
  cantEncimaDelPromedio(){
    let promedio = this.edadPromedio(),
        cant = 0
    this.estuds.forEach((estud)=>{
      if (estud.edad > promedio) cant++
    })
    return cant
  }
}
```

Se cargan los datos

Al crear el objeto salón, se inicializa el array en vacío. Se van cargando los estudiantes al ARRAY **estuds**.

Para obtener los que están encima del promedio

Primero será preciso **CALCULAR LA EDAD PROMEDIO**.

El método edadPromedio **retorna un NÚMERO**

El método CantEncimaDelPromedio

Primero guarda el valor **promedio**. Luego vuelve a recorrer el array para ver si el objeto está por encima del promedio, y lo cuenta de ser así.

Este método **retorna un NÚMERO**

Ver demo: 0. Ejemplo Salón de Clases

La estructura de datos ARRAY

Así hemos trabajado hasta ahora: leemos un objeto y de inmediato procesamos los datos de procesos universales.

Ahora el objeto Cl_salon contiene un array con Cl_estud, y lo vamos cargando dinámicamente (agregar / eliminar). Al requerir un resultado, **se recorre el arreglo para calcularlo**.

estuds

nombre: Luis edad: 18
nombre: Ana edad: 20
nombre: Eva edad: 16
nombre: Ray edad: 20

- Ahora los objetos se guardan TODOS dentro del ARRAY
- El ARRAY tiene varias posiciones, comenzando en 0
- Cada posición se accede con corchetes. Ej. estuds[0]
- Al referirnos a **estuds**, es todo el arreglo.
- Al referirnos a una posición, ej. **Estuds[1]**, es un objeto
- Al referirnos a un componente, Ej. **Estuds[2]**.edad accedemos al atributo o método del objeto

edadPromedio = SE RECORRE EL ARRAY

- ✓ En un método de la clase Cl_salon
- ✓ acumEdad y cntEstuds serán variables locales al método

```
class Cl_salon{
  :::::::
  quienesEncimaPromedio(){
    let prom = this.edadPromedio()
    return this.estuds.filter(e=>
      e.edad > prom)
  }
  edadMayor(){
    let mayor = 0
    this.estuds.forEach(e=>{
      if (e.edad > mayor) mayor = e.edad
    })
    return mayor
  }
  quienesEdadMayor(){
    let mayor = this.edadMayor()
    return
      this.estuds.filter(e => e.edad===mayor)
  }
  quienesConEdad(edad){
    return
      this.estuds.filter(e => e.edad===edad)
  }
}
```

El método quienesEncimaPromedio
Primero guarda el valor **promedio**.
Luego filtra los registros con edad mayor al promedio.
RETORNA un arreglo con los que cumplen la condición
Este método **retorna un ARRAY**

El método edadMayor
Recorre el arreglo para determinar la edad mayor
Este método **retorna un NÚMERO**

Método quienesEdadMayor
Filtra los de edad mayor
Este método **retorna un ARRAY**

Método quienesConEdad
Filtra según **la edad dada**
Este método **retorna un ARRAY**

Ver demo: O. Ejemplo Salón de Clases

```
edadPromedio(){
  let acumEdad = 0, cntEstuds = 0
  this.estuds.forEach((estud)=>{
    acumEdad += estud.edad
    cntEstuds++
  })
  return acumEdad / cntEstuds
}
```

edadPromedio

- ✓ Inicia variables locales acumEdad y cntEstuds (proceso universal **PROMEDIO**)
- ✓ Recorre el array con forEach, acumulando edades y contando estudiantes
- ✓ Retorna proceso universal **PROMEDIO**: acumEdad / cntEstuds (**UN NÚMERO**)

Retorna **UN NÚMERO**: **edadPromedio = 18,5**

estuds

nombre: Luis edad: 18
nombre: Ana edad: 20
nombre: Eva edad: 16
nombre: Ray edad: 20

cantEncimaDelPromedio

- ✓ Obtiene el promedio (del método anterior)
- ✓ Inicializa la variable local cant, para contar los que están encima del promedio (proceso universal **CONTAR**)
- ✓ Recorre el array con forEach y compara cada edad a ver si está encima del promedio
- ✓ Retorna el proceso universal **CONTAR**: cant (**UN NÚMERO**)

```
cantEncimaDelPromedio(){
  let promedio = this.edadPromedio(),
      cant = 0
  this.estuds.forEach((estud)=>{
    if (estud.edad > promedio) cant++
  })
  return cant
}
```

Retorna **UN NÚMERO**: **cantEncimaDelPromedio = 2**

Ver demo: 0. Ejemplo Salón de Clases

```
quienesEncimaPromedio(){
  let prom = this.edadPromedio()
  return this.estuds.filter(e=>
    e.edad > prom)
}
```

quienesEncimaPromedio

- ✓ Obtiene el promedio (del método anterior)
- ✓ Recorre el array con **filter**, para obtener los objetos con estudiantes con edad mayor al promedio
- ✓ Retorna **UN ARRAY** con los elementos del arreglo que cumplen la condición (**UN ARRAY**)

Retorna **UN ARRAY**: **quienesEncimaPromedio**

nombre: Ana edad: 20
nombre: Ray edad: 20

estuds

nombre: Luis edad: 18
nombre: Ana edad: 20
nombre: Eva edad: 16
nombre: Ray edad: 20

edadMayor

- ✓ Inicializa la variable local **mayor**, para el proceso de mayor (proceso universal **MAYOR**)
- ✓ Recorre el array con forEach y compara cada edad a ver si es mayor que **mayor**.
- ✓ Retorna el proceso universal **MAYOR**: mayor (**UN NÚMERO**)

```
edadMayor(){
  let mayor = 0
  this.estuds.forEach(e=>{
    if (e.edad > mayor) mayor = e.edad
  })
  return mayor
}
```

Retorna **UN NÚMERO**: **edadMayor = 20**

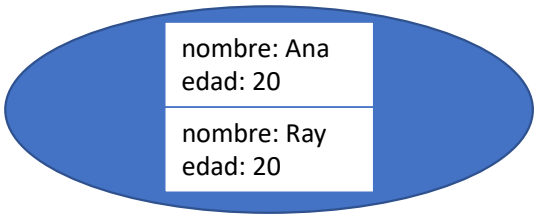
Ver demo: 0. Ejemplo Salón de Clases

```
quienesEdadMayor(){
  let mayor = this.edadMayor()
  return
    this.estuds.filter(e => e.edad===mayor)
}
```

quienesEdadMayor

- ✓ Obtiene la edad mayor (del método anterior)
- ✓ Recorre el array con **filter**, para obtener los objetos con estudiantes con edad igual a edadMayor
- ✓ Retorna **UN ARRAY** con los elementos del arreglo que cumplen la condición (**UN ARRAY**)

Retorna **UN ARRAY**: **quienesEdadMayor**



estuds

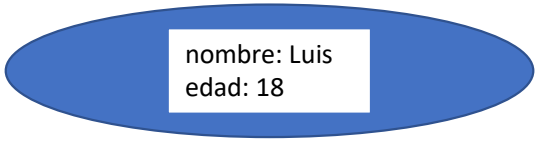
nombre: Luis edad: 18
nombre: Ana edad: 20
nombre: Eva edad: 16
nombre: Ray edad: 20

quienesConEdad

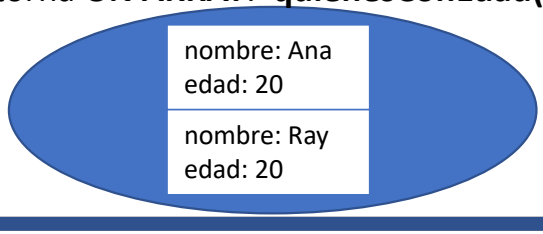
- ✓ Recorre el array con filter y compara cada edad a ver si es igual a la que viene por argumento (parámetro).
- ✓ Retorna **UN ARRAY** con los objetos que cumplen la condición

```
quienesConEdad(edad){
  return
    this.estuds.filter(e => e.edad===edad)
}
```

Ej.1. Retorna **UN ARRAY**: **quienesConEdad(18)**



Ej.2. Retorna **UN ARRAY**: **quienesConEdad(20)**



Ver demo: O. Ejemplo Salón de Clases

Más ejemplos del uso de Arrays

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

El objeto Array de JavaScript es un objeto global que es usado en la construcción de arrays, que son objetos tipo lista de alto nivel.

Los arrays son objetos similares a una lista cuyo prototipo proporciona métodos para efectuar operaciones de recorrido y de mutación. Tanto la longitud como el tipo de los elementos de un array son variables. Dado que la longitud de un array puede cambiar en cualquier momento, y los datos se pueden almacenar en ubicaciones no contiguas, no hay garantía de que los arrays de JavaScript sean densos.

Operaciones habituales

Crear un Array

```JS
let frutas = ["Manzana", "Banana"];

console.log(frutas.length);
// 2
```

Acceder a un elemento de Array mediante su índice

```JS
let primero = frutas[0];
// Manzana

let ultimo = frutas[frutas.length - 1];
// Banana
```

Recorrer un Array

```JS
frutas.forEach(function (elemento, indice, array) {
  console.log(elemento, indice);
});
// Manzana 0
// Banana 1
```

Añadir un elemento al final de un Array

```JS
let nuevaLongitud = frutas.push("Naranja"); // Añade "Naranja" al final
// ["Manzana", "Banana", "Naranja"]
```

Eliminar el último elemento de un Array

```JS
let ultimo = frutas.pop(); // Elimina "Naranja" del final
// ["Manzana", "Banana"]
```

Añadir un elemento al principio de un Array

```JS
let nuevaLongitud = frutas.unshift("Fresa"); // Añade "Fresa" al inicio
// ["Fresa" ,"Manzana", "Banana"]
```

Eliminar el primer elemento de un Array

```JS
let primero = frutas.shift(); // Elimina "Fresa" del inicio
// ["Manzana", "Banana"]
```

Más ejemplos del uso de Arrays

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

Encontrar el índice de un elemento del Array

```
JS

frutas.push("Fresa");
// ["Manzana", "Banana", "Fresa"]

let pos = frutas.indexOf("Banana"); // (pos) es la posición para abreviar
// 1
```

Eliminar un único elemento mediante su posición

Ejemplo:

Eliminamos "Banana" del *array* pasándole dos parámetros: la posición del primer elemento que se elimina y el número de elementos que queremos eliminar. De esta forma, `.splice(pos, 1)` empieza en la posición que nos indica el valor de la variable `pos` y elimina 1 elemento. En este caso, como `pos` vale 1, elimina un elemento comenzando en la posición 1 del *array*, es decir "Banana".

```
JS

let elementoEliminado = frutas.splice(pos, 1);
// ["Manzana", "Fresa"]
```

mdn web docs_

Eliminar varios elementos a partir de una posición

Nota:

Con `.splice()` no solo se puede eliminar elementos del array, si no que también podemos extraerlos guardándolo en un nuevo array. ¡Ojo! que al hacer esto estaríamos modificando el array de origen.

```
JS

let vegetales = ["Repollo", "Nabo", "Rábano", "Zanahoria"];
console.log(vegetales);
// ["Repollo", "Nabo", "Rábano", "Zanahoria"]

let pos = 1,
  numElementos = 2;

let elementosEliminados = vegetales.splice(pos, numElementos);
// ["Nabo", "Rábano"] ==> Lo que se ha guardado en "elementosEliminados"

console.log(vegetales);
// ["Repollo", "Zanahoria"] ==> Lo que actualmente tiene "vegetales"
```

Copiar un Array

```
JS

let copiaArray = vegetales.slice();
// ["Repollo", "Zanahoria"]; ==> Copiado en "copiaArray"
```

Más ejemplos del uso de Arrays

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

Acceso a elementos de un array

Los índices de los *arrays* de JavaScript comienzan en cero, es decir, el índice del primer elemento de un *array* es `0`, y el del último elemento es igual al valor de la propiedad `length` del *array* restándole 1.

Si se utiliza un número de índice no válido, se obtendrá `undefined`.

JS

```
let arr = [
  "este es el primer elemento",
  "este es el segundo elemento",
  "este es el último elemento",
];
console.log(arr[0]); // escribe en consola 'este es el primer elemento'
console.log(arr[1]); // escribe en consola 'este es el segundo elemento'
console.log(arr[arr.length - 1]); // escribe en consola 'este es el último elemento'
```

Los elementos de un *array* pueden considerarse propiedades del objeto tanto como `toString` (sin embargo, para ser precisos, `toString()` es un método). Sin embargo, se obtendrá un error de sintaxis si se intenta acceder a un elemento de un *array* de la forma siguiente, ya que el nombre de la propiedad no sería válido:

JS

```
console.log(arr.0) // error de sintaxis
```

No hay nada especial ni en los *arrays* de JavaScript ni en sus propiedades que ocasione esto. En JavaScript, las propiedades cuyo nombre comienza con un dígito no pueden referenciarse con la notación punto y debe accederse a ellas mediante la notación corchete.

Por ejemplo, dado un objeto con una propiedad de nombre `'3d'`, sólo podría accederse a dicha propiedad con la notación corchete.

JS

```
let decadas = [1950, 1960, 1970, 1980, 1990, 2000, 2010]
console.log(decadas.0)  // error de sintaxis
console.log(decadas[0]) // funciona correctamente
```

JS

```
renderizador.3d.usarTextura(modelo, 'personaje.png')
renderizador['3d'].usarTextura(modelo, 'personaje.png')
```

Obsérvese que, en el último ejemplo, ha sido necesario poner `'3d'` entre comillas. Es posible usar también comillas con los índices del los *arrays* de JavaScript (p. ej., `decadas['2']` en vez de `decadas[2]`), aunque no es necesario.

El motor de JavaScript transforma en un string el 2 de `decadas[2]` a través de una conversión implícita mediante `toString`. Por tanto, `'2'` y `'02'` harían referencia a dos posiciones diferentes en el objeto `decadas`, y el siguiente ejemplo podría dar `true` como resultado:

JS

```
console.log(decadas["2"] != decadas["02"]);
```

Más ejemplos del uso de Arrays

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

Relación entre `length` y las propiedades numéricas

La propiedad `length` de un *array* de JavaScript está conectada con algunas otras de sus propiedades numéricas.

Varios de los métodos propios de un *array* (p. ej., `join()`, `slice()`, `indexOf()`, etc.) tienen en cuenta el valor de la propiedad `length` de un array cuando se les llama.

Otros métodos (p. ej., `push()`, `splice()`, etc.) modifican la propiedad `length` de un array.

```
JS

const frutas = [];
frutas.push("banana", "manzana", "pera");

console.log(frutas.length); // 3
```

mdn web docs_

Cuando se le da a una propiedad de un *array* JavaScript un valor que corresponda a un índice válido para el *array* pero que se encuentre fuera de sus límites, el motor actualizará el valor de la propiedad `length` como corresponda:

```
JS

frutas[5] = "fresa";
console.log(frutas[5]); // 'fresa'
console.log(Object.keys(frutas)); // ['0', '1', '2', '5']
console.log(frutas.length); // 6
```

Si se aumenta el valor de `length`:

```
JS

frutas.length = 10;
console.log(frutas); // ['banana', 'manzana', 'pera', <2 empty items>, 'fresa', <4 empty items>]
console.log(Object.keys(frutas)); // ['0', '1', '2', '5']
console.log(frutas.length); // 10
console.log(frutas[8]); // undefined
```

Si se disminuye el valor de la propiedad `length` pueden eliminarse elementos:

```
JS

frutas.length = 2;
console.log(Object.keys(frutas)); // ['0', '1']
console.log(frutas.length); // 2
```