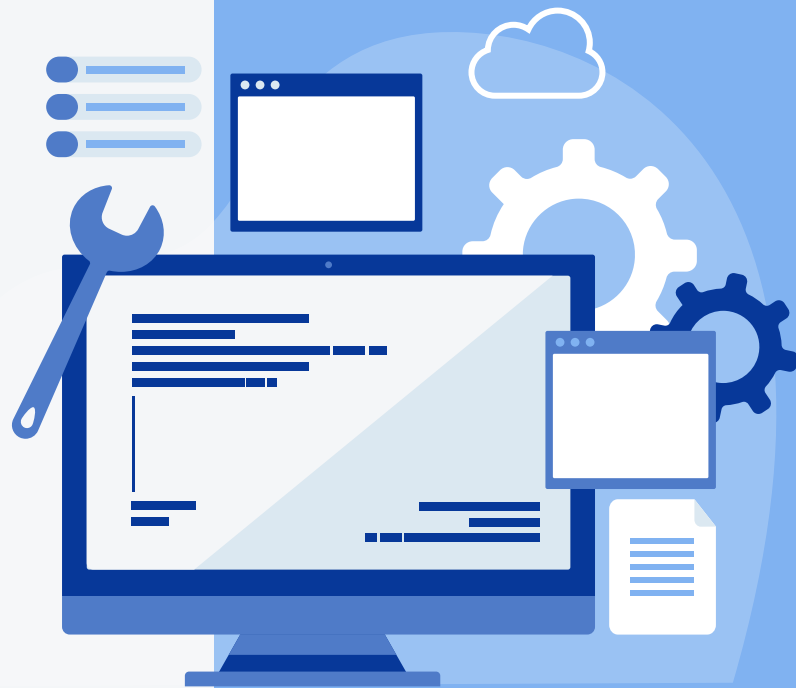


Objetos y arreglos en JS

Cordero Hernández, Marco R.



En sesiones pasadas...



- Introducción a JS
- Funciones
 - Expresiones de funciones
 - Callbacks
 - Arrow functions

CONTENIDOS

01

Objetos

02

Arreglos



01

Objetos



Objetos

JS puede almacenar datos complejos (estructuras de datos abstractas), a lo cual se le conocerá como *objeto*.

```
let obj1 = new Object(); // Constructor tradicional
let obj2 = {}; // Sintaxis de objeto literal

let estudiante = {
  nombre: 'Ricardo', // Propiedad nombre con valor 'Ricardo'
  carrera: 'ICOM',
  'apellido paterno': 'Hernández'
  // Clave con varias palabras van entre comillas
}
```



Propiedades de objetos

```
// Acceso a propiedades
console.log(estudiante.nombre); // Ricardo
console.log(estudiante['nombre']); // Ricardo
console.log(estudiante['apellido paterno']); // Hernández

// Seteo de nuevas propiedades
estudiante.promedio = 9.85;
console.log(estudiante.promedio); // 9.85

// Eliminación de propiedades
delete estudiante.carrera;
console.log(estudiante.carrera); // ¿Error?

// Acceso dinámico al objeto
let ap = 'apellido paterno';
estudiante[ap] = 'Fernández';
console.log(estudiante[ap]); // Fernández
```



Flexibilidad de declaración

```
let exp = 'expediente';
let estudiante2 = {
  [exp]: 727272, // Equivalente a expediente: 727272
  nombre: 'Hassan', // Pueden haber comas al final
};
console.log(estudiante2.expediente); // 727272

// Uso de palabras reservadas dentro de objetos
let valido = {
  let: 'variable',
  for: 'ciclo',
  return: 'retorno de función'
};
console.log(valido.for); // ciclo
```



Función generadora

```
// Función generadora
function nuevoEstudiante(nombre, carrera) {
    return {
        nombre: nombre,
        carrera: carrera
    };
}


// ... Equivalente a
function nuevoEstudiante(nombre, carrera) {
    return {
        nombre, carrera
    };
}
```


01.1

Objetos - Constructores

Concepto de objetos: Constructor

Así como muchos lenguajes de programación orientados a objetos, **JS** cuenta con la capacidad de definir atributos, métodos y demás. Bajo esta premisa, es posible declarar el *constructor* de una clase.

Object	Properties	Methods
	<code>car.name = Fiat</code> <code>car.model = 500</code> <code>car.weight = 850kg</code> <code>car.color = white</code>	<code>car.start()</code> <code>car.drive()</code> <code>car.brake()</code> <code>car.stop()</code>



Uso de constructor

```
// Método primitivo
function Auto(color, año) {
    this.color = color
    this._año = año;
}

let jetta = new Auto('plata', 2006);

// Uso de clases
class Auto {
    #año;
    constructor(color, año) {
        this.color = color;
        this.#año = año;
    }
};

let vento = new Auto('cafe', 2015);
```



Existencia de propiedades

```
let auto = new Auto('azul', 2024);

// Revisión de propiedades
if (auto.asientos === undefined) {
    console.log('Objeto auto no tiene esta propiedad' );
}

if (!('asientos' in auto)) {
    console.log('Objeto auto no tiene esta propiedad' );
}

// Recorrido del objeto
for (let key in auto) {
    console.log(`${key}: ${auto[key]}`);
} // ¿Qué imprimirá?
```

01.2

Objetos - Comparación



Comparación y clonación

```
let auto1 = new Auto('negro', 2000, 42_000);
let auto2 = auto1;
auto2.km = 0;
console.log(auto1.km); // ?

// Alternativa (Object.assign(target, source1, source2, ...))
Object.assign(auto2, auto1);
auto2.km = 42_000;
console.log(auto2);
console.log(auto1.km); // ?

// Comparativas
if (auto1 == auto2) { /* ¿Entrará? */ }
if (auto1 === auto2) { /* ¿Entrará? */ }

let a = {}, b = {};
if (a == b) { /* ¿Entrará? */ }
if (a === b) { /* ¿Entrará? */ }
```

Ejercicio

- Crear un objeto con la información de un usuario (uid, nombre, apellidos, email, password, fecha, sexo, imagen)
 - Usa <https://randomuser.me/api/portraits/men/0.jpg> para las imágenes
- Implementa una función generadora de usuarios (con el método resumido)
- Implementa la clase *User* y muestra todas las propiedades del usuario
- Crea una función que indique si un objeto recibido tiene o no alguna propiedad, y de tenerla, mostrar el valor de esa propiedad

01.3

Objetos - Getters/Setters



Uso de la referencia “this”

En OOP, es común que un objeto necesite acceder a la información contenida dentro de sus atributos en tiempo de ejecución para algún propósito. Para lograr esto, se usa la palabra clave **this**.

```
let artista = {  
  nombre: 'MF DOOM',  
  edad: 49,  
  
  // Equivalente a mostrarNombre: function () {...}  
  mostrarNombre() {  
    console.log(this.nombre);  
    // Equivalente a...  
    console.log(artista.nombre);  
  }  
};
```



Uso de la referencia “this”

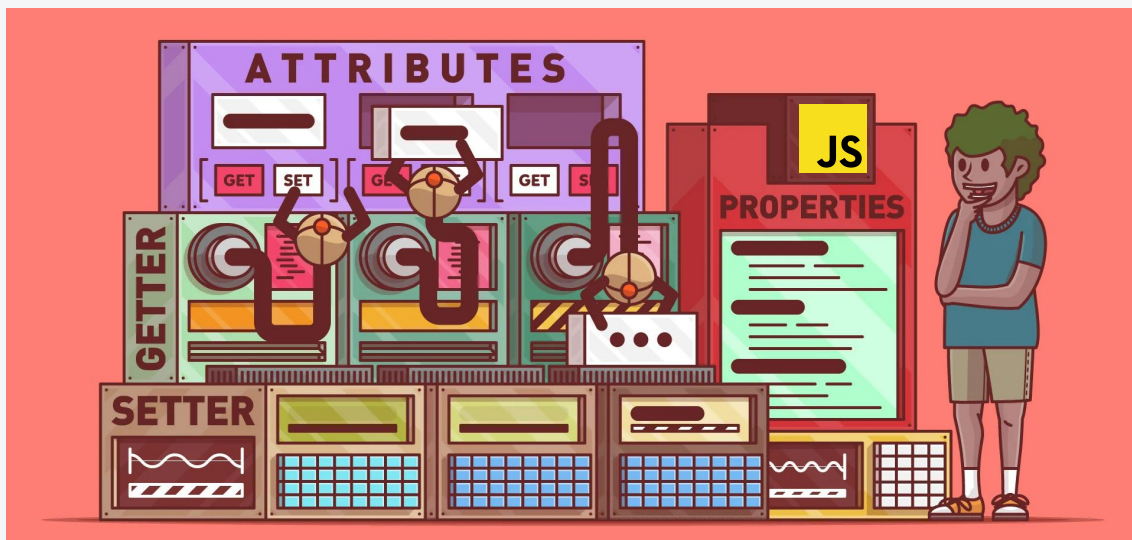
```
// Muestra del nombre
artista.mostrarNombre(); // ?

// Sobrescritura
let otroArtista = artista;
artista = null;
otroArtista.mostrarNombre(); // ?

// Asignación de métodos
let artista2 = { nombre: 'Lng/SHT' };
function mostrarNombre() {
    console.log(this);
}
artista2.otraFuncion = mostrarNombre;
artista2.otraFuncion(); // ?
artista2['otraFuncion'](); // ?
mostrarNombre(); // ?
```

Getters y Setters

Haciendo referencia nuevamente a OOP, es posible crear *getters* y *setters* al menos de manera *virtual*, es decir, no son parte de atributos originales pero pueden ser una composición





Getters y Setters

```
let alumno = {
  nombre: 'Pedro Genérico',
  calificacion: 5.9,

  get reporte() {
    return `${this.nombre} -> ${this.calificacion >= 6 ? "a" : "re"}probado`;
  },

  set reporte(valores) {
    let [nombre, calificacion] = valores.split(" ");
    this.nombre = nombre;
    this.calificacion = Number(calificacion);
  }
};

console.log(alumno.reporte); // Uso de getter -> ?
alumno.reporte = 'Marco 8.35';
console.log(alumno.reporte); // ?
```



Getters y Setters

El uso de getters y setters dentro de una clase requiere de un manejo adicional cuando se desee acceder a las propiedades que afecta.

```
class Alumno {  
    constructor(nombre, carrera) {  
        this.nombre = nombre;  
        this.carrera = carrera;  
    }  
  
    get nombre() {  
        return this._nombre;  
    }  
  
    set nombre(val) {  
        this._nombre = val  
    }  
}
```

Los atributos “verdaderos” serán nombrados con un guión bajo al inicio, ej.: `this._nombre`.

P: ¿Qué pasaría si se utiliza `this.nombre` dentro de un setter?

01.4

Objetos – Función flecha

Funciones flechas (arrow functions)



Las funciones flecha no cuentan con su propio **this**, más bien hacen referencia al *contexto* donde se encuentran o la función externa de donde se declaran.

```
// Correcto
let artista3 = {
  nombre: 'Peso Lápiz',
  mostrarNombre() {
    let mostrar = () => console.log(this.nombre);
    mostrar();
  }
};
artista3.mostrarNombre(); // ?

// Incorrecto
let artista4 = {
  nombre: 'Peso Plumón',
  mostrarNombre() {
    let mostrar = function () { console.log(this.nombre); };
    mostrar();
  }
};
artista4.mostrarNombre(); // ?
```



Argumentos

El objeto especial ***arguments*** describe los argumentos de la función en donde se utiliza.

Las funciones flecha no pueden obtener sus argumentos, más bien toma los argumentos de la función en donde se encuentra contenida.

```
function crearArtista(nombre, genero) {  
    console.log(arguments);  
}  
crearArtista('Frank Ocean', 'Dance');  
  
function conFlecha(prueba) {  
    let crearOtroArtista = (nombre, genero) => {  
        console.log(arguments);  
    };  
    crearOtroArtista('(José)²', 'Regional');  
}  
conFlecha('prueba');
```


Ejercicio

Añadir ***getters*** y ***setters*** al ejercicio previo (clase de usuarios).

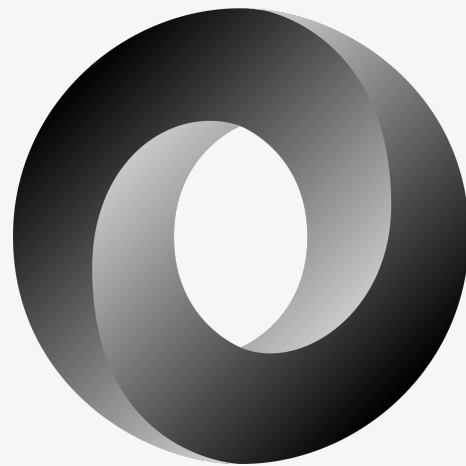
01.5

Objetos - JSON

JSON

Un archivo **JSON** es un formato para intercambio de datos, el cual puede ser interpretado fácilmente por humanos y por computadoras, ya que su sintaxis es sencilla en cuestión de legibilidad y flexible en cuestión de generación.

Evidentemente, **JS** tiene múltiples formas de manipular estos documentos de forma nativa.





Uso de JSON

```
let persona = {  
  nombre: 'Antonio',  
  edad: 57,  
  idiomas: {  
    español: '100%',  
    inglés: '43%',  
    latin: '0%',  
    italiano: '75%'  
  },  
  hobbies: [  
    'fútbol americano',  
    'ajedrez', 'patear niños'  
  ]  
};
```

```
// Visualización del objeto nativo  
console.log(persona);  
  
// Manejo de JSON  
let json = JSON.stringify(persona);  
  
// Visualización del objeto en formato  
JSON  
console.log(json);  
console.log(typeof json); // ?  
  
// Conversión de JSON a objeto nativo  
let obj = JSON.parse(json);  
console.log(obj.edad); // ?  
console.log(typeof obj); // ?
```



Datos permitidos en JSON

De manera nativa, JS soporta la conversión de {objetos}, [arreglos] y primitivas (String, Number, Boolean y null).

```
// Soporte integrado y conversión en línea  
  
let strJSON = '{"nombre":"Juanito","nacimiento":"2001-05-23T12:00:00.000Z"}' ;  
  
let obj2 = JSON.parse(strJSON, function(k, v) {  
    return (k == 'nacimiento') ? new Date(v) : v;  
});  
  
console.log(obj2.nacimiento.getFullYear()); // ?
```



02

Arreglos



Declaración de arreglos

Similar a otros lenguajes de programación, existen múltiples formas de inicializar arreglos.

```
let arr1 = new Array(); // Constructor
let arr2 = []; // Forma tradicional

let arr3 = new Array(5); // ?
console.log(arr3[0]); // ?
arr3.push(0); // ?
console.log(arr3[6]); // ?

let arr4 = [1, 2, 3, 4]; // Asignación inmediata
let arr5 = new Array(1, 2, 3, 4, 5); // Equivalente

console.log(arr5.length); // ?
```



Iteración de arreglos

Al contar con la longitud del arreglo, es posible saber dónde inicia y dónde termina, por consecuencia, se puede recorrer de inicio a fin.

```
let arr1 = [1, 2, 5, 20, -99, 2000];

// Con for tradicional
for (let i = 0; i < arr1.length; i++) {
  console.log(arr1[i]);
}

// Con for-each
for (let i of arr1) {
  console.log(i);
}
```




Longitud de arreglos

Todos los arreglos cuentan con la propiedad nativa ***length*** para saber conocer su longitud, la cual puede ser modificada en el momento que se requiera.

```
let arr1 = [3, 4, 5, 1, 2, 10];  
console.log(arr1.length); // ?  
arr1.length = 3;  
console.log(arr1); // ?  
arr1.length = 6;  
console.log(arr1[4]); // ?  
  
// Limpiar el arreglo  
arr1.length = 0;
```



Matrices

Haciendo alusión al concepto matemático, JS soporta *matrices*, es decir, arreglos dentro de otros arreglos con **N** niveles de anidación.

```
let mat1 = [  
  [1, 2],  
  [3, 4],  
  [5, 6]  
];  
console.log(mat1[2][0]); // ?
```



Concatenación (y más magia de JS)

Los arreglos también pueden ser concatenados con otros objetos, sin embargo, demuestran un comportamiento singular.

```
console.log([] + 2); // ?  
console.log([2] + 2); // ?  
console.log([2, 3] + 2); // ?
```

Ejercicio

- Crea un arreglo de usuarios (***const users = []***)
- Agrega tres usuarios (***users.push(...)***)
- Muestra el arreglo usando ***console.table*** y también conviértelo a ***JSON***
- Cambia la longitud del arreglo a “1” y vuelve a mostrar el arreglo

02.1

Arreglos - Métodos principales

Métodos de arreglos básicos

Al ser implementados de manera abstracta, los arreglos requieren de métodos de alto nivel para su modificación y adición. En JS, los siguientes atributos y métodos pueden ser empleados:

- *length* - Tamaño del arreglo
- ***push()*** - Inserta al final del arreglo
- ***unshift()*** - Inserta *elementos* al inicio del arreglo
- ***pop()*** - Elimina y retorna el último elemento del arreglo
- ***shift()*** - Elimina y retorna el primer elemento del arreglo



Métodos de arreglos básicos

```
let arr = ['uno', {  
  nombre: 'Juan'  
}, true, function () {  
  console.log('Hola mundo');  
}];
```

```
console.log(arr[1].nombre); // Acceso a objeto  
arr[3](); // Llamada a función  
console.log(arr.length); // ?  
console.log(arr); // Impresión del arreglo
```



Métodos de arreglos básicos

```
let arr2 = [1, 2, 3, ]; // Sintaxis válida

console.log(arr2); // Vista inicial
arr2.push(4);
arr2.push(5, 6);
arr2.shift(); // ?
arr2.pop(); // ?
arr2.unshift(-1);
arr2.unshift(-3, -2);
console.log(arr2); // ?
```


Ejercicio

- Crea una función ***addUser*** que recibe los atributos de un nuevo usuario, le asigna un “uid” y lo añade al arreglo
- Asegúrate de que no exista otro usuario con el mismo nombre y apellidos, ni con el mismo correo
- Puedes asignarle una imagen de forma automática con la información del género y el id % 100 para construir la url (ej. uid = 153 → [https://randomuser.me/api/portraits/\\${genero}/\\${uid%100}.jpg](https://randomuser.me/api/portraits/${genero}/${uid%100}.jpg))

02.2

Arreglos - Métodos adicionales

Métodos adicionales

Los siguientes métodos también están disponibles:

- *delete* - Limpiar el valor de un elemento en una posición *i* al asignarlo con valor *undefined* (`delete arr[i]`)
- ***splice(index[, deleteCount, e1, e2, ..., eN])*** - Recorre los elementos y elimina o inserta según el modo de uso
- ***slice(start, end)*** - Regresa un nuevo arreglo conteniendo los elementos del inicio al fin *sin incluir el fin*
- ***concat(a1, a2, ..., aN)*** - Une un arreglo con otros y regresa la unión resultante
- ***forEach(function(item, index, array){})*** - Recorre todos los elementos del arreglo y ejecuta alguna acción en cada uno



delete

Limpiar el valor de un elemento y lo asigna a undefined.

```
// Sintaxis -> delete arr[index]
let array1 = ['uno', 'dos', "tres"];
delete array1[1];
console.log(array1); // ?
console.log(array1.length); // ?
```



Array.prototype.splice

Elimina o inserta elementos en la posición indicada y *devuelve lo que se sustituirá*.

```
// Sintaxis -> arr.splice(start index[, deleteCount, e1, e2, ..., eN])
array1.splice(1, 1); // Desde index 1 eliminar 1 elemento
console.log(array1); // ['uno', 'tres']
array1.splice(2, 0, 'cuatro', 'cinco'); // En el índice 2 inserta 2 elementos
console.log(array1); // ['uno', 'tres', 'cuatro', 'cinco']
array1.splice(-1, 0, "v1", "v2"); // Inserta en la penúltima posición
console.log(array1); // ['uno', 'tres', 'cuatro', 'v1', 'v2', 'cinco']
console.log(array1.splice(2)); // Regresa ['cuatro', 'v1', 'v2', 'cinco']
console.log(array1); // ['uno', 'tres']
```



Array.prototype.slice

Regresa un nuevo subarreglo de inicio a fin (exclusivo por la derecha) [inicio, fin).

```
// Sintaxis -> arr.slice(start, end)
let array2 = [1, 2, 3, 5, 8, 13, 21]
console.log(array2.slice(1, 5)); // [2, 3, 5, 8]
console.log(array2); // ?
```



Array.prototype.concat

Une varios arreglos hacía un objetivo y regresa uno nuevo con la unión total.

```
// Sintaxis -> arr.concat(arr1, arr2, ..., arrN)
let array3 = array2.concat(array2, array1);
console.log(array3);
/**
    [
      1,   2,   3,   5,
      8,  13, 21,   1,
      2,   3,   5,   8,
      13, 21, 'uno', 'tres'
    ]
 */
```



Array.prototype.forEach

Para evitar usar las estructuras de bloque `for` y `for-each`, el método (ahora) nativo de los arreglos *forEach* puede ser usado para recorrer todos los elementos y realizar acciones sobre los mismos.

```
// Sintaxis -> arr.forEach(function(item, index, array) {...})
array3.forEach(function(item, index, array) {
    console.log(item, index, array);
});

array3.forEach((item, index, array) => { // Función flecha
    console.log(item);
});

array3.forEach(item => console.log(item)); // Solo los elementos
```


02.3

Arreglos - Búsquedas y filtros

Métodos *indexOf*, *find* y *filter*

Para buscar y filtrar elementos dentro de un arreglo, se cuenta con los siguientes métodos:

- ***indexOf(item, start)*** - Busca un elemento dentro del arreglo desde un punto de partida (índice)
- ***lastIndexOf(item, start)*** - Busca la última instancia dentro del arreglo
- ***includes(item, start)*** - Determina si un elemento está dentro de un arreglo
- ***find(criteria)*** - Busca un elemento dentro del arreglo con criterios personalizados
- ***findIndex(criteria)*** - Devuelve el índice del primer elemento que cumpla con los criterios de búsqueda (o -1 si no encuentra nada)
- ***filter(criteria)*** - Devuelve un nuevo arreglo con los elementos filtrados según el criterio definido



Métodos *indexOf*, *find* y *filter*

```
let array = [1, 2, 3, 2, 1, 5];

// Index of
// Sintaxis -> arr.indexOf(item, start)
console.log(array.indexOf(2, 0)); // 1
console.log(array.indexOf(1, 1)); // ?

// Last Index Of
// Sintaxis -> arr.lastIndexOf(item, start)
console.log(array.lastIndexOf(1)); // 4

// Includes
// Sintaxis -> arr.includes(item, from)
console.log(array.includes(4, 0)); // false
console.log(array.includes(3, 0)); // true
```



Métodos *indexOf*, *find* y *filter*

```
let users = [
  {id: 1, nombre: 'Jesús'}, {id: 2, nombre: 'María'},
  {id: 3, nombre: 'Jaime'}
];

// Find
// Sintaxis -> arr.find(function)
let mUser = users.find(item => item.nombre.startsWith('M'));
console.log(mUser); // ?
console.log(users.find(user => user.id == 4)); // ?

// Find Index
// Sintaxis -> arr.findIndex(function)
let nonUser = users.findIndex(u => u.id == 32);
console.log(nonUser); // -1

// Filter
// Sintaxis -> arr.filter(function)
let jUsers = users.filter(usrs => usrs.nombre.startsWith('J'));
console.log(jUsers, jUsers.length); // ?
```

Ejercicio

- Crea una función ***updateUser*** que reciba un id y un objeto con los valores a actualizar; verifica que exista el usuario y copia los valores *excepto el (u)id y el correo* (estos no tendrán que modificarse)
 - Usa ***findIndex*** y ***Object.assign***
- Crea otra función ***deleteUser*** que reciba un id y verifique si existe el usuario correspondiente, en ese caso, borrarlo
 - Usa ***findIndex*** y ***splice***

02.4

Arreglos - Conceptos extra



Métodos extendidos

```
let names = ['Eliud', 'Alfredito', 'Chalino'];

// Map
// Sintaxis -> arr.map(function)
// Propósito -> "Mapear" un valor a otro
let tamaños = names.map(nombre => nombre.length);
console.log(tamaños); // [5, 9, 7]

// Sort
// Sintaxis -> arr.sort([function])
// Propósito -> Ordenar un arreglo
tamaños.sort();
console.log(tamaños); // [5, 7, 9]

names.sort((a, b) => {
  if (a.toUpperCase() < b.toUpperCase()) return -1;
  if (a.toUpperCase() > b.toUpperCase()) return 1;
  return 0;
});
console.log(names); // ['Alfredito', 'Chalino', 'Eliud']
```



Métodos extendidos

```
// Join
// Sintaxis -> arr.join('sepString')
// Propósito -> Unir los elementos de un arreglo
let joined = names.join(', ');
console.log(joined); // 'Alfredito, Chalino, Eliud'

// Reduce
// Sintaxis -> arr.reduce(function, initialValue)
// Propósito -> Reducir un arreglo al acumularlo
let reduced = tamaños.reduce((a, b) => a + b);
console.log(reduced); // 21
let prefix = tamaños.reduce((a, b) => a + b, 42);
console.log(prefix); // 63

// Is Array
// Sintaxis -> Array.isArray(object)
// Propósito -> Determinar si un objeto es un arreglo
console.log(Array.isArray([])); // true
console.log(Array.isArray({})); // false

// Todos los métodos disponibles con
// Object.getOwnPropertyNames(Array.prototype)
```


Desestructuración (Destructuring)

JS permite distribuir los elementos de un arreglo y almacenarlos en múltiples variables, todo a la vez en una sola línea. A esto se le conoce como “desestructuración”, ya que se toma la estructura de una arreglo y se “parte” para manejarla como si sus elementos fueran independientes.

```
let arrNames = ['Jorge', 'Barbosa']
let [nombre, apellido] = arrNames;
console.log(nombre); // Jorge
console.log(apellido); // Barbosa

let [val1, val2] = "Hola,mundo,cruel".split(',');
console.log(val1); // ?
console.log(val2); // ?

let [, , tercero] = [1, 2, 1000, -99_000];
console.log(tercero); // ?
```

Ejercicio

- Crea un arreglo y llénalo con 10 usuarios usando un *for*
 - Llenando los ids incrementalmente
 - Cambia el nombre del usuario por “Nombre 0”, “Nombre 1”, ..., “Nombre 9”
 - Cambia el apellido y el correo del usuario por “Apellido 0”, “correo0@gmail.com”, ...
 - De manera **random** genera una imagen para el usuario entre 0 y 100
 - De manera **random** elige el género del usuario
- Crea una función **sortUsers(cb)** que recibe una función como argumento (callback) que indica cómo hacer el ordenamiento
 - Internamente usa el método **sort** para el arreglo de usuarios que ejecuta ese callback

Ejercicio

- Crea 2 funciones para ordenar por apellido (alfabéticamente) y ordenar por correo de manera descendente
 - Utiliza **sortUsers** para mandar a llamar ambas funciones
- Mostrar con un **forEach** el formato
 - uid - correo
- Mostrar los usuarios en una sola línea con el formato
 - 0->Nombre 0, 1->Nombre 1, ...