Gestión de arrays y datos complejos

Un array, en cualquier lenguaje de programación, es una variable que sirve para almacenar varios datos dentro. En muchos lenguajes estos arrays tienen un tamaño fijo e inamovible durante la ejecución del programa, pero en JavaScript no (de hecho, no son arrays propiamente dichos, sino vectores o listas).

Normalmente los datos que se almacenan en un array son del mismo tipo (por ejemplo, un conjunto de números enteros, o un conjunto de cadenas de texto), aunque en el caso de JavaScript y otros lenguajes (como PHP), cada elemento de la lista o conjunto puede ser de un tipo diferente al resto.

1. Manipulación básica de arrays en JavaScript

Para crear arrays en JavaScript, creamos la variable que va a almacenar el conjunto con dos corchetes o bien con `new Array`. En este último caso, podemos indicar el tamaño inicial del array también:

```
let conjunto = [];
let conjunto2 = new Array();
let conjunto3 = new Array(10);
```

También podemos definir a la vez el array y los elementos iniciales que va a tener en una misma línea (aunque luego podemos añadir más o cambiar los que hay).

```
let conjunto = [10, "Hola", 3.5];
let conjunto2 = new Array(10, "Hola", 3.5);
```

Después, podemos ir añadiendo elementos a esta colección usando los corchetes y poniendo dentro qué posición ocupa el elemento en la colección, empezando por la posición 0. Por ejemplo:

```
conjunto[0] = 10;
conjunto[1] = "Hola";
conjunto[2] = 3.5;
...
```

Alternativamente, también podemos emplear la instrucción `push` para añadir elementos al final de los existentes:

```
conjunto.push(3);
conjunto.push("Buenas");
```

Observemos cómo cada posición guarda un dato distinto (un entero, un texto, un número real...). Esto es posible, como decimos, en ciertos lenguajes de programación, especialmente los de tipado dinámico, es decir, aquellos donde no hay que indicar de qué tipo es una variable, sino que toma el tipo del valor que se le asigna. Si volvemos a utilizar una misma posición, borraremos el elemento previo y asignaremos el nuevo, pudiendo también cambiar el tipo de dato de esa posición:

```
conjunto[1] = "Adiós";  // Ahora este elemento ya no vale "Hola"
```

Si utilizamos una posición no correlativa con las anteriores, se añadirá un elemento en esa posición, y quedarán huecos vacíos enmedio, a los que se les asigna el valor *undefined*.

1.1. Arrays y referencias

Si asignamos a una variable simple el valor de otra, estamos creando una copia de la variable original, con lo que si modificamos la segunda variable, la primera no ve su valor alterado:

```
let a = 3;
let b = a;
b = 4;
alert(a);       // 3
```

Esto no es así con los arrays, ya que almacenan un conjunto mayor de datos. Cuando asignamos una variable array a otra, estamos asignando una referencia a la posición de memoria del array original, con lo que cualquier cambio en esa segunda variable va a afectar al array original:

```
let datos = [1, 2, 3, 4];
let datos2 = datos;
datos2[0] = 10;
console.log(datos);         // [10, 2, 3, 4]
```

2. Recorrido de arrays

En ocasiones es posible que nos toque recorrer una lista o array de elementos de una web, buscando uno en concreto. Para ello, disponemos de las estructuras repetitivas o bucles vistos en documentos anteriores (*while*, *do..while* o *for*). Por ejemplo, supongamos que tenemos una variable *lista* con un array de elementos (el que

sea). Si queremos recorrerlopodemos utilizar una variable que vaya desde el principio del array (posición 0) hasta el final (indicado por la propiedad `length` del array):

```
for (let i = 0; i < lista.length; i++)
{
    alert (lista[i]);   // Muestra el valor de cada elemento
}
```

Otra alternativa es utilizar una variable que sirva de índice en la lista, tomando el valor de cada posición de la misma, y accediendo con dicho valor a la posición concreta del array:

```
for (let indice in lista)
{
    alert (lista[indice]);  // Muestra el valor de cada elemento
}
```

Una tercera alternativa que podemos emplear desde ES2015 es el bucle *for..of*, que permite iterar por todos los valores del array (y también por todas las letras de un texto):

```
for (let valor of lista)
{
    alert(valor);
}
```

Sobre la propiedad `length`, se cuentan las posiciones hasta la última ocupada, aunque en medio haya posiciones no definidas (*undefined*). Por ejemplo, este array tiene tamaño 5:

```
let a = [];
a[4] = 3;
```

3. Operaciones con arrays

Pasamos a detallar ahora algunas de las operaciones más útiles que podemos realizar con arrays:

- Ya hemos visto que la instrucción `push` nos sirve para añadir elementos al final de los disponibles en un array. Podemos pasarle tantos como queramos, separados por comas. La instrucción `unshift` añade elementos al principio del array. También podemos pasarle los que queramos, separados por comas.

- La función `pop` devuelve y elimina el último elemento del array. La función `shift` devuelve y elimina la primera posición.
- La función `join` se aplica sobre un array. Puede recibir como parámetro un delimitador (texto) y obtiene un texto separando todos los elementos del array con el delimitador.
- La función `concat` enlaza dos arrays, obteniendo otro con la unión de todos los elementos de ambos
- La función `slice` obtiene un subarray desde una posición inicial (incluida) hasta una final (no incluida)
- La función `splice` quita elementos de un array, indicando desde qué posición eliminar y cuántos elementos quitar desde ahí. Adicionalmente, admite más parámetros, que serían los datos a añadir en la posición donde nos hemos quedado.
- La función `reverse` invierte el orden de los elementos de un array
- La función `indexOf` recibe como parámetro un dato, y nos devuelve en qué posición del array se encuentra por primera vez, o -1 si no se encuentra.
- La función `includes` recibe como parámetro un dato y devuelve si dicho dato está en el array o no.

Veamos aquí un ejemplo de uso de cada una de estas funciones:

```
let datos = [1, 2, 3, 4];
datos.unshift(5, 6);                // [5, 6, 1, 2, 3, 4]
let variable = datos.pop();         // variable = 4, datos = [5, 6, 1, 2, 3]
let texto = datos.join(":");        // "5:6:1:2:3"
let datos2 = datos.concat([10, 9]); // [5, 6, 1, 2, 3, 10, 9]
let datos3 = datos.slice(1, 3);     // [6, 1]
datos2.splice(1, 3, 20, 30);        // [5, 20, 30, 3, 10, 9]
datos2.reverse();                   // [9, 10, 3, 30, 20, 5]
let posicion = datos2.indexOf(3);   // 2
let existe = datos2.includes(10);   // true
```

> **Ejercicio 1**:
>
> Realiza los siguientes pasos en un fichero llamado **arrays_js.html**. Muestra por consola el resultado después de aplicar cada uno, pero con los elementos separados por "=>" usando *join*:
>
> - Crea un array con 4 elementos numéricos
> - Concatena 2 elementos más al final y 2 al principio (*unshift* y *push*)
> - Elimina las posiciones de la 3 a la 5 (incluida) (*splice*)
> - Inserta 2 elementos más entre el penúltimo y el último (*splice*)
> - Invierte el array (*reverse*)

3.1. Ordenación de arrays

La función `sort` permite ordenar el array sobre el que se aplica. Dicha ordenación suele ser alfabética (por ejemplo, 40 < 5, o "Adiós" < "Hola").

```
let nombres = ["Juan", "Ana", "Laura", "Blas"];
nombres.sort();     // ["Ana", "Blas", "Juan", "Laura"]
```

Sin embargo, podemos pasar como parámetro a la función sort una función anónima que defina un criterio de comparación alternativo. Dicha función recibe dos parámetros que actúan como dos datos cualesquiera del array, y devuelve un número que será:

- Negativo si el primer parámetro debe ir antes que el segundo
- Positivo si el primer parámetro debe ir después que el segundo
- Cero si ambos parámetros son iguales, y deben ir uno junto al otro en la ordenación

El siguiente ejemplo ordena ascendentemente (en orden numérico) un array de enteros:

```
let datos = [20, 4, 6, 2, 10];
datos.sort(function(n1, n2) {
    if (n1 < n2)
        return -1;
    else if (n1 > n2)
        return 1;
    else
        return 0;
});
```

Alternativamente, podemos expresarla de esta otra forma más resumida, con el mismo resultado:

```
let datos = [20, 4, 6, 2, 10];
datos.sort(function(n1, n2) {
    return n1 - n2;
});
```

> **Ejercicio 2**:
>
> Crea un fichero llamado **arrays_ordenacion_js.html**. Crea dentro un array de números y muéstralo ordenado de mayor a menor.

3.2. Otras operaciones avanzadas con arrays

Existen otras operaciones que podemos aplicar sobre arrays que tienen un uso un poco más avanzado. Muchas de ellas reciben una función como parámetro, que se aplica a cada elemento del array.

La función `every` obtiene un booleano indicando si la función que recibe como parámetro es cierta para todos los elementos del array. El siguiente ejemplo comprueba si todos los elementos del array son pares:

```
let resultado = datos.every(function(n) {
    return n % 2 == 0;
});
```

La función `some` es similar a la anterior, pero devuelve un booleano indicando si alguno de los elementos del array cumple la función. Este ejemplo comprueba si alguno de los números del array es mayor que 10:

```
let resultado = datos.some(function(n) {
    return n > 10;
});
```

La función `forEach` recorre cada elemento del array, aplicándole la función que recibe como parámetro. Este ejemplo muestra por consola cada elemento del array:

```
datos.forEach(function(n) {
    console.log(n);
});
```

La función `filter` obtiene como resultado un array con los elementos que cumplen la condición indicada por la función que se pasa como parámetro. El siguiente ejemplo obtiene un subconjunto del array inicial con los datos que sean mayores que 10:

```
let datosMayores10 = datos.filter(function(n){
    return n > 10;
});
```

La función `map` permite aplicar una transformación (definida por la función recibida como parámetro) a todos los elementos de un array. El siguiente ejemplo multiplica por 2 cada elemento del array:

```
let dobles = datos.map(function(n) {
    return n * 2;
});
```

Notar que en todas estas funciones podemos emplear *arrow functions* o funciones lambda como parámetro:

```
let resultado = datos.every(n => n % 2 == 0);
let datosMayores10 = datos.filter(n => n > 10);
let dobles = datos.map(n => n * 2);
```

> **Ejercicio 3**:
>
> Crea un fichero llamado **funciones_arrays_js.html**. Define una función que reciba un array y un número como parámetros y devuelva si todos los números del array son múltiplos del número recibido como parámetro. Haz que el número tome por defecto el valor de 2, y prueba la función con distintas llamadas.

> **Ejercicio 4**:
>
> Crea un fichero llamado **funciones_arrays2_js.html**. Define un array de datos numéricos, quédate sólo con los números que sean positivos y transforma el array para que almacene los cuadrados de esos números. Usa funciones lambda para resolver este ejercicio.

4. Objetos JavaScript

Aunque JavaScript no es un lenguaje orientado a objetos propiamente dicho, sí que permite definir clases (veremos algún ejemplo de ello más adelante) y trabajar con objetos que almacenan información heterogénea.

Para definir un objeto JavaScript, encerramos sus datos entre llaves. Cada propiedad que queramos definir llevará un nombre y, después de los dos puntos, le podemos dar un valor inicial. Así podríamos definir los datos de una persona:

```
let persona = {
    nombre: "Nacho",
    edad: 43,
    telefono: "611223344"
};

console.log(persona.nombre);
```

Como podemos comprobar, una vez hemos definido los datos de la persona u objeto, accedemos a ellos con el operador punto `.`, separando el nombre de la variable y el del campo a consultar. También podemos añadir sobre la marcha nuevos campos sobre el objeto ya creado:

```
let persona = {
    nombre: "Nacho",
    edad: 43,
    telefono: "611223344"
};
persona.direccion = "C/Mayor 13";
```

Ejercicio 5:

Crea una página llamada **objetos_js.html**. Define un array con datos de personas como los del ejemplo anterior. Ordena después el array ascendentemente por el nombre de las personas (alfabético), recórrelo y muéstralo por pantalla. Puede serte de utilidad la función *localeCompare* que permite comparar alfabéticamente dos textos y obtener un entero indicando cuál es mayor o menor.