Arrays in Javascript

Arrays are a special kind of object and data structure where one name can store multiple values.

Example:

```
const array_x = [374, 755, 398, 118, 3, 27]
```

Each individual value in an array is called an **array element.**
Each array element has an index position. Index positions start at 0 for the first one and count forward in integers for the following position.

For the example above, the **array element index positions** are:
[0] = 374, [1] = 755. [2] = 398, [4] = 118, [5] = 3, [6] = 27.

The element index values are used for access of a value from within an array.

```
array_x[3];
array_x[5] + array_x[6];
array_x[1] – array_x[0];
array_x[2] * array_x[5];
array_x[0] / array_x[4];
```

- Arrays can have varying datatypes stored within them.
- ANY datatype in Javascript can be used as an element within an array.
- Datatypes may be mixed within the element set of an array.

text strings as a datatype:

```
shoppingList = ["bread", "milk", "cheese", "hummus", "noodles"]
colors = ["red", "green", "blue"]
```

Arrays may be declared as variables or constants, using the let or const keywords, or less commonly, the Array constructor.

Array Literals:

An **array litera**l is the simplest and most common way to create an array by enclosing a comma-separated list of elements within square brackets ([]). **This is generally the preferred method over using the Array() constructor.**

```
const newNumbers = [3,5,7,1,2,4,6];
```

The other common way is to declare/initialize the array using the **Array Constructor**:

```
const games = new Array('dark_souls', 'darkSouls3', 'bloodborne');
console.log(games);
console.log(games[1]);
```

Declaring an array as a variable:

Arrays may be declared as variables.

- `const system_ID = [001, 002, 003, 004];`
- `let testGroupZ = [susan, david, mary, jill, alan];`
- `const fruits = new Array("Pineapple", "Banana", "Apple");`

Arrays may be declared as empty of element values, or with values:

```
let myArray = new Array();
let numbers = new Array(10, 20, 30);
```

Arrays, similar to variables, have declaration and initialization, which may be done either together or separately.

```
const formulaZ = [9, 8, 7, 6, 5, 3,1];
let formulaX = [];
```

Printing out an individual value from an array:

console.log can be used as a printout method fairly easily. The array must already exist.

```
const formulaZ = [9, 8, 7, 6, 5, 3,1];
console.log(formulaZ[3]);    //THIS PRINTS OUT THE VALUE 6
```

Either the entire set of an array may be printed out, or individual values, or any combination thereof:

```
const numberSet = [1, 3, 5, 7, 9, 2, 4, 6, 8];

console.log(numberSet);

console.log(numberSet[0], numberSet[2], numberSet[4]);
```

The length method: counts the length of element values in the array.

```
const numberSet = [1, 3, 5, 7, 9, 2, 4, 6, 8];
let z = numberSet.length;

console.log(z);
```

Basic Array Methods:

METHODS FOR MANIPULATING VALUES WITHIN THE ARRAY:

1. **push** – adds a value to the end of an array

```
const numberSet = [1, 3, 5, 7, 9, 2, 4, 6, 8];
numberSet.push(7);
console.log(numberSet);
```

2. **pop** – removes the end value

```
const numberSet = [1, 3, 5, 7, 9, 2, 4, 6, 8];
numberSet.pop();
console.log(numberSet);
```

3. **unshift** – adds a value to front of the array

```
const numberSet = [1, 3, 5, 7, 9, 2, 4, 6, 8];
numberSet.unshift(0);
console.log(numberSet);
```

4. **shift** – remove an array value from the front of the array

```
const numberSet = [1, 3, 5, 7, 9, 2, 4, 6, 8];
numberSet.shift();
console.log(numberSet);
```

5. **reverse** – reverses order of values in the array

```
const numberSet = [1, 3, 5, 7, 9, 2, 4, 6, 8];
numberSet.reverse();
console.log(numberSet);
```

METHODS FOR USING THE ARRAY VALUES:

6. **includes** – gives a false or true boolean response based on whether or not a value is included in an array

```
const numberSet = [1, 3, 5, 7, 9, 2, 4, 6, 8];

let z = numberSet.includes(9);

console.log(z);
```

7. **indexOf** – shows the array index of a value

```
const numberSet = [1, 3, 5, 7, 9, 2, 4, 6, 8];

let z = numberSet.indexOf(9);

console.log(z);
```

8. **slice** – lists array values including and after a given index value

```
const numbers2 = [34, 55, 95, 20, 15];

let n = numbers2.slice(2);

console.log(n);
```

slice can also be done in a range:

```
n = numbers2.slice(2,4);

console.log(n);
```

9. **splice**

The Array.prototype.splice() method in JavaScript is a versatile and powerful tool that **changes the contents of an array by removing, replacing, or adding elements in place**, and it **mutates the original array**.

The basic syntax for the splice() method is:

```
array.splice(startIndex, deleteCount, item1, item2, ...);
```

The parameters for splice() include **startIndex**, the required index to start changing the array; an optional **deleteCount** for the number of elements to remove; and optional **item1, item2,** ... to add elements at the starting index. The method returns an array of the removed elements.

splice() supports **removing**, **adding**, and **replacing** elements.

To remove elements, you specify startIndex and deleteCount.

Adding elements involves setting deleteCount to 0 and providing items.

Replacing elements requires providing startIndex, deleteCount for removal, and the new items.

It's important to distinguish splice() from slice(). While splice() modifies the original array, slice() creates a new array.

Example:

```
const fruits = ['apple', 'banana', 'orange', 'grape'];

fruits.splice(1, 2, 'pear', 'kiwi');

console.log(fruits);

// Result: ['apple', 'pear', 'kiwi', 'grape']
```

using splice to change just one item: (removes single item)

```
const number_b = [34, 55, 95, 20, 15];

let b = number_b.splice(3, 1);

console.log(b);
```

slice() does not modify the original array, it simply uses it as a source.

splice() actually modifies the array in place.

some general notes on arrays:

- arrays can be used as elements in arrays – they can be nested
- arrays can be concatenated
- arrays may use the spread operator

Example of a simple nested array:

```
const grid = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];
```

This array has 3 arrays nested within it; each grouping is an array.

concatenation is the joining of two separate arrays.

Example of a concatenated array:

```
const arr1 = ['apple', 'banana'];

const arr2 = ['cherry', 'date'];

// Concatenate arr1 and arr2

const combinedArray = arr1.concat(arr2);



console.log(combinedArray); // Output: ['apple', 'banana', 'cherry', 'date']

console.log(arr1);          // Output: ['apple', 'banana'] (original
unchanged)
```

Arrays in Javascript may use the Spread Operator (…).

The JavaScript **spread operator** (...) . It is commonly used to make copies, combine data, and pass arguments to functions.

Key Uses and Examples

- **Copying Arrays/Objects:** The spread operator creates a new shallow copy of an existing array or object, which is important for maintaining immutability in modern JavaScript development (e.g., in frameworks like React).

```
const originalArray = [1, 2, 3];
const copiedArray = [...originalArray];
// copiedArray is [1, 2, 3], a new array instance

const originalObject = { name: 'Alice', age: 30 };
const copiedObject = { ...originalObject };
// copiedObject is { name: 'Alice', age: 30 }
```

- **Combining/Merging Arrays/Objects:** It simplifies combining multiple iterables or objects into a single new one without using methods like concat() or Object.assign().

```
const numbersOne = [1, 2, 3];
const numbersTwo = [4, 5, 6];
const combinedNumbers = [...numbersOne, ...numbersTwo];
// combinedNumbers is [1, 2, 3, 4, 5, 6]

const car = { brand: 'Ford', model: 'Mustang' };
const carDetails = { type: 'car', year: 2021, color: 'yellow' };
const myCar = { ...car, ...carDetails };
// myCar is { brand: 'Ford', model: 'Mustang', type: 'car', year: 2021,
color: 'yellow' }
```

If properties have the same name when merging objects, the value from the last object in the spread operation will overwrite earlier ones.

- **Passing Arguments to Functions:** The spread operator expands an array of values into individual arguments for a function call, useful for functions that expect a variable number of parameters.

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

// It is also commonly used with built-in functions like Math.max()
const values = [10, 4, 20, 5];
console.log(Math.max(...values)); // Output: 20
```

- **Converting Iterables to Arrays:** It can easily convert other iterables, such as strings, into an array where each element is a character.

```
const str = 'Hello';
const strArr = [...str];
// strArr is ['H', 'e', 'l', 'l', 'o']
```

Spread vs. Rest Operator

Both the spread operator and the **rest operator** use the same ... syntax, but they serve opposite purposes.

- **Spread** *expands* an iterable into individual elements. It is used in expression contexts, like in array or object literals or function calls.
- **Rest** *gathers* multiple elements or properties into a single array or object. It is used in assignment contexts, such as in function parameters or array/object destructuring.