**How to Shuffle an Array of Items Using JavaScript or TypeScript**

[Grant Weatherston](https://www.freecodecamp.org/news/author/gweaths/)



In this article we'll be exploring how we can shuffle an array of items in multiple different ways using TypeScript, or JavaScript should you prefer.

Pre-Requisites:

* An understanding of TypeScript or JavaScript
* A basic understanding of For Loops and Arrays

The following examples are written in TypeScript, but they work in exactly the same way using plain JavaScript. You just simply need to remove the :type syntax from all function parameters.

**Method 1: Fisher-Yates Sorting Algorithm**

This algorithm's basic premise is to iterate over the items, swapping each element in the array with a randomly selected element from the remaining un-shuffled portion of the array.

Let's take a look at this in practice, as it'll help you visualise it better:

// declare the function

const shuffle = (array: string[]) => {

for (let i = array.length - 1; i > 0; i--) {

const j = Math.floor(Math.random() \* (i + 1));

[array[i], array[j]] = [array[j], array[i]];

}

return array;

};

// Usage

const myArray = ["apple", "banana", "cherry", "date", "elderberry"];

const shuffledArray = shuffle(myArray);

console.log(shuffledArray);

**Explanation**

First, you create a for loop. This will allow you to loop over each item in the array, swapping its position with another item in the array.

You then create the i variable assigning it the value of length of the array - 1.

You do this because we're starting at the last element of the array, and all array indexes start at 0 – so the last index would be 4 (as there are 5 items in the array).

If you were to try and access myArray[i] with i equalling 5 (the length) it would throw an exception stating there is no item at index 5. So we subtract 1 from the length.

By starting from the last element and working your way backwards, you guarantee that elements towards the end of the array have an equal chance of being swapped with any other element.

If you were to shuffle the array from the beginning to the end, the elements towards the beginning of the array would have a higher chance of being swapped multiple times, leading to a biased or uneven shuffle.‌‌‌‌ You then create a j variable which will be used for your index pointer for the big swap.

You then assign the array at index i to the array at index j, and visa versa. This swaps the values and shuffles them up for each item in the array.

**Array Destructuring Assignment Explained**

The syntax [array[i], array[j]] = [array[j], array[i]] is called an *array destructuring assignment*. It allows for the swapping of values between two variables or array elements without the need for a temporary variable.

Here's how the array destructuring assignment works in the context of shuffling an array using the Fisher-Yates shuffle algorithm:

* array[i] and array[j] represent two elements in the array that need to be swapped.
* [array[j], array[i]] creates a temporary array containing the values of array[j] and array[i], but in reverse order.
* By assigning [array[j], array[i]] to [array[i], array[j]], the values of array[j] and array[i] are swapped in place.

You can learn more about array destructuring [here](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment).‌‌‌‌ If we were to print out the steps of the function step by step we would would get the following:

// starting array

["apple", "banana", "cherry", "date", "elderberry"];

//1st Iteration - Swap elderberry with date

i = 4; // elderberry

j = 3; // date

[("apple", "banana", "cherry", "elderberry", "date")];

// 2nd Iteration - Swap elderberry with apple

i = 3; //

elderberry j = 0; // apple

[("elderberry", "banana", "cherry", "apple", "date")];

// 3rd Iteration - Swap cherry with banana

i = 2; // cherry

j = 1; // banana

[("elderberry", "cherry", "banana", "apple", "date")];

// 4th Iteration - Swap cherry with itself (stays where it is)

i = 1; // cherry

j = 1; // cherry

[("elderberry", "cherry", "banana", "apple", "date")];

And there you have it – we have shuffled the array using the Fisher-Yates algorithm.

**Method 2: Using the sort() Method with a Random Comparison Function**

If you're not familiar with the JS sort function, I've written a tutorial on how to use this which you can find [here](https://www.freecodecamp.org/news/how-does-the-javascript-sort-function-work/).

const shuffle = (array: string[]) => {

return array.sort(() => Math.random() - 0.5);

};

// Usage

const myArray = ["apple", "banana", "cherry", "date", "elderberry"];

const shuffledArray = shuffle(myArray);

console.log(shuffledArray);

This is a simple sorting() function that returns a random number, which would either work out as a negative, 0, or positive number.‌‌‌‌The sort() method internally compares pairs of elements in the array and determines their relative order based on the return value of the comparison function.

* If the comparison function returns a negative value, the first element is considered smaller and should be placed before the second element in the sorted array.
* If the comparison function returns a positive value, the first element is considered larger and should be placed after the second element in the sorted array
* If the comparison function returns 0, the relative order of the elements remains unchanged.

**What does Math.random() return?**

When you call Math.random() it generates a pseudo-random number, because as you may know, nothing is *really* random haha.‌‌‌‌*"Pseudo-random"* means that the numbers generated appear to be random but are actually determined by a deterministic algorithm (which differs between JS engine implementations).‌‌‌‌The number it returns will always be a floating number, between 0 and 1. Floating numbers (commonly referred to as "floats") are numbers which can be positive or negative and can have a fractional part. Examples of floating-point numbers include *3.14, -0.5, 1.0, 2.71828*, and so on.

**Why do you subtract 0.5 from the result of Math.random()?**

By subtracting 0.5 from the result of Math.random(), you introduce a random value between -0.5 and 0.5. This random value will cause the comparison function to return negative, positive, or zero values in a random manner for different pairs of elements. Consequently, the sort() method shuffles the array randomly.

**Method 3: Using the JS Array.map() Function**

The .map() function allows you to iterate over each element of an array and transform them into new values based on a provided mapping function. The map() function returns a new array with the transformed values, leaving the original array unchanged.

const shuffle = (array: string[]) => {

return array.map((a) => ({ sort: Math.random(), value: a }))

.sort((a, b) => a.sort - b.sort)

.map((a) => a.value);

};

// Usage

const myArray = ["apple", "banana", "cherry", "date","elderberry"]; const shuffledArray = shuffle(myArray);

console.log(shuffledArray);

Here, you loop over the array and use the same Math.random() function as you did in the above example within the map() function, returning an array of the objects with a sort number, and a value.‌‌‌‌You can then use the sort() function to sort the array based on these values, before calling the map() function again to create an array of values (that is, the string names).

const shuffle = (array: string[]) => {

const shuffled = array.slice();

let currentIndex = shuffled.length;

let temporaryValue, randomIndex;

while (currentIndex !== 0) {

randomIndex = Math.floor(Math.random() \* currentIndex);

currentIndex -= 1;

temporaryValue = shuffled[currentIndex];

shuffled[currentIndex] = shuffled[randomIndex];

shuffled[randomIndex] = temporaryValue;

}

return shuffled;

};

// Usage

const myArray = ["apple", "banana", "cherry", "date", "elderberry"]; const shuffledArray = shuffle(myArray);

console.log(shuffledArray);

**Conclusion**

I'd recommend the Fisher-Yates algorithm method, as it is has a low time complexity, as its all dependent on the size of the array. Its time complexity can be seen as *O(n)*, which implies that doubling the input size will roughly double the execution time. Similarly, if the input size is halved, the execution time will be approximately halved as well.

Without going into too much detail:

* O represents the order of growth or complexity class.
* n refers to the size of the input, typically represented by the variable n.

Meaning that the complexity of the function changes as the n (input) changes in size – for example: Complexity x 2 (2 items in array) vs Complexity x 10 (10 items in array). So the larger the array the larger the complexity and time taken to shuffle.

This is worth noting when shuffling large arrays. It could be worth looking at other methods, or perhaps chunking the array and running the shuffling in parallel before piecing it back together.

This method also allows for easier shuffling of any type of array, not just string[] too. It would also work very nicely when using TypeScript generics. This allows for any type of array to be passed to the function, and shuffled.

const shuffle = <T>(array: T[]) => {

for (let i = array.length - 1; i > 0; i--) {

const j = Math.floor(Math.random() \* (i + 1));

[array[i], array[j]] = [array[j], array[i]];

}

return array;

};

const strings = ["apple", "banana", "cherry", "date", "elderberry"];

const users = [ { name: "John", surname: "Doe" }, { name: "Jane", surname: "Doe" }];

const shuffledArray = shuffle(strings);

const shuffledObjects = shuffle(users);

console.log(shuffledArray);

console.log(shuffledObjects);

I hope you this found this article useful, and have learnt how to easily shuffle an array of items. You could use this for multiple use cases such as:

* Shuffle a playlist of songs
* Shuffling a list of team members to determine a rota system
* Creating a quiz of random questions / order-randomising data samples, for example customer reviews / feedback.

As always feel free to [follow me o](http://twitter.com/gweaths)