JavaScript Array sort() Method Overview

The following is a summary glossing over how to use the JavaScript array prototype `sort()` method to reorganize element positions within arrays of objects, strings, and numbers. The elements from the original inputted array will have their positions rearranged by the `sort()` method, which returns a sorted array. Using the `sort()` method, elements will be sorted in ascending order, from smallest to largest value, by default. The `sort()` method transforms array elements into strings before it performs a value comparison to determine the sorting order of the returned array.

```
Ex.
Input:
let nums = [0, 1 , 2, 3, 10, 20, 30];

numbers.sort();

console.log(numbers);


Output:
[0, 1, 10, 2, 20, 3, 30]
```

As we can observe in this example, the `sort()` method reorders `10` before `2` because the string `'10'` is ordered before `'2'` when comparing the two strings against each other. To remedy this default behavior, a `compare()` function must be passed to the `sort()` method so that it can determine the correct order of elements.

```
Ex.
Correct syntax for the sort() method:
array.sort(compareFunction);
```

Without including the compare function, the `sort()` method will place the elements of the array in order based on string comparison. The `sort()` method's compare function accepts two arguments then returns the value which is used to decide the order of the elements.

```
Ex.
Correct syntax for the compare() function:
function compare(a,b) {}
```

In the above example, `a` and `b` are the two arguments accepted by the `compare()` function. The value returned by the `compare()` function will force the `sort()` method to arrange elements based on the following stipulations:

- `compare(a, b) < 0` The sort method will place `a` at the lowest index and `b` will have the greatest index.

- `compare(a, b) > 0` The sort method will place `b` at the lowest index and `a` will have the greatest index.
- `compare(a, b) === 0` The sort method determines `a` is equivalent to `b` and their index remains unchanged.

```
Ex.
Input:
let numbers = [0, 1, 2, 3, 10, 20, 30];

numbers.sort((a, b) => {
    if (a > b) return 1;

    if (a < b) return -1;

    return 0;
});

console.log(numbers);


Output:
[0, 1, 2, 3, 10, 20, 30]
```

To simplify things further, the `sort()` method with a `compare()` function can be included in one line seeing as it's only sorting an array of numbers at the moment.

```
Ex.
Input:
let numbers = [0, 1, 2, 3, 10, 20, 30];

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

console.log(numbers);


Output:
[0, 1, 2, 3, 10, 20, 30]
```

Sorting an Array of Strings

```
Ex.
let animals = ['cat',
               'dog',
               'elephant',
               'bee',
               'ant'];
```

The default behavior of the `sort()` method can be used to arrange the elements of the animals array in ascending alphabetical order without having to worry about passing the `compare()` function.

```
Ex.
Input:
let animals = ['cat',
               'dog',
               'elephant',
               'bee',
               'ant'];

animals.sort();

console.log(animals);


Output:
['ant',
 'bee',
 'cat',
 'dog',
 'elephant']
```

This array is also able to be arranged in descending alphabetical order by configuring the `compare()` function appropriately, as demonstrated in the following example.

```
Ex.
Input:
let animals = ['cat',
               'dog',
               'elephant',
               'bee',
               'ant'];
```

Example continues on next page...

```
animals.sort((a, b) => {
    if (a > b) return -1;

    if (a < b) return 1;

    return 0;
});

console.log(animals);


Output:
['elephant',
 'dog',
 'cat',
 'bee',
 'ant']
```

We can also modify arguments passed to the `compare()` function in order to sort elements which may contain a combination of upper and lowercase characters inside the animals array. In the following example, this is accomplished by converting all elements to the same case prior to comparison by customizing the `compare()` function's arguments with the `.toUpperCase()` method before passing the `compare()` function to the `sort()` method.

```
Ex.
Input:
let mixedCaseAnimals = ['Cat',
                        'dog',
                        'Elephant',
                        'bee',
                        'ant'];
```

Example continues on next page...

```
mixedCaseAnimals.sort((a, b) => {
    let x = a.toUpperCase(),
        y = b.toUpperCase();

    return x == y ? 0 : x > y ? 1 : -1;
});


Output:
['ant',
 'bee',
 'Cat',
 'dog',
 'Elephant']
```

Sorting an Array of Strings with Non-ASCII Characters

Strings that contain non-ASCII characters (é, è, etc...) are not able to be rearranged correctly using the default `sort()` method. In order to remedy this situation, a specific locale is able to be considered by using the `localeCompare()` method on any strings that contain non-ASCII characters.

```
Ex.
let animaux = ['zèbre',
               'abeille',
               'écureuil',
               'chat'];
```

Example continues on next page...

```
animaux.sort();

console.log(animaux);
```

In the above example of an array of French animal species names, the `zébre` string is meant to come after the `écureuil` string once the sort method is applied. To accomplish correct sorting for this example, we can deploy the `localeCompare()` method like this:

```
Ex.
Input:
animaux.sort((a, b) => {
    return a.localeCompare(b);
});

console.log(animaux);


Output:
['abeille',
 'chat',
 'écureuil',
 'zèbre']
```

Sorting an Array of Numbers

```
Ex.
let scores = [9, 80, 10, 20, 5, 70];
```

A custom `compare()` function can be used to govern the way the `sort()` method reorders an array of numbers. This example illustrates how to ascendingly sort a numerical array:

```
Ex.
Input:
let scores = [9, 80, 10, 20, 5, 70];

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

console.log(scores);


Output:
[5, 9, 10, 20, 70, 80]
```

If we reverse the logic in the `compare()` function of the above example, we are able to accomplish descending order sorting of a numerical array:

```
Ex.
Input:
let scores = [9, 80, 10, 20, 5, 70];

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

console.log(scores);


Output:
[80, 70, 20, 10, 9, 5]
```

Sorting an Array of Objects by a Specified Property

```
Ex.
let employees = [
    {name: 'Josie',
     salary: 90000,
     hireDate: 'August 27, 2023'},
    {name: 'Olivia',
     salary: 75000,
     hireDate: 'December 13, 2022'},
    {name: 'Britney',
     salary: 80000,
     hireDate: 'January 1, 2024'},
];
```

In this example we have an array of three objects which each contain three different types of properties. The `name` is a string, the `salary` is a number, and the `hireDate` is a date.

Sorting an Array of Objects by a Numeric Property

The employee objects can be sorted in ascending order based on their numeric `salary` property by doing the following:

```
Ex.
Input:
employees.sort((x, y) => {
    return x.salary - y.salary;
});

console.table(employees);


Output: [
    {name: 'Olivia',
     salary: 75000,
     hireDate: 'December 13, 2022'},
    {name: 'Britney',
     salary: 80000,
     hireDate: 'January 1, 2024'},
    {name: 'Josie',
     salary: 90000,
     hireDate: 'August 27, 2023'},
]
```

The above example shares similarities with our approach to ascendingly arranging an array of numbers, because here we're comparing the numeric `salary` property to establish an ascending sort order for the `employee` objects.

Sorting an Array of Objects by a String Property

When sorting the `employee` objects by the string `name` property, we can deploy the same sort of solution as the one we used before when comparing and reordering case-insensitive strings in an array.

```
Ex.
Input:
employees.sort((x, y) => {
    let a = x.name.toUpperCase(),
        b = y.name.toUpperCase();

    return a == b ? 0 : a > b ? 1 : -1;
});

console.table(employees);


Output: [
    {name: 'Britney',
     salary: 80000,
     hireDate: 'January 1, 2024'},
    {name: 'Olivia',
     salary: 75000,
     hireDate: 'December 13, 2022'},
    {name: 'Josie',
     salary: 90000,
     hireDate: 'August 27, 2023'},
]
```

Sorting an Array of Objects by a Date Property

To sort the `employee` objects by their individual hire dates, we must first convert the `hireDate` property strings into `Date` objects so they can be properly compared against each other numerically.

```
Ex.
Input:
employees.sort((x, y) => {
    let a = new Date(x.hireDate),
        b = new Date(y.hireDate);

    return a - b;
});

console.table(employees);


Output: [
    {name: 'Olivia',
     salary: 75000,
     hireDate: 'December 13, 2022'},
    {name: 'Josie',
     salary: 90000,
     hireDate: 'August 27, 2023'},
    {name: 'Britney',
     salary: 80000,
     hireDate: 'January 1, 2024'},
]
```

Optimizing the JavaScript Array sort() Method

It's important to keep in mind that the `compare()` function is called numerous times by the `sort()` method for each array element.

```
Ex.
Input:
let rivers = ['Nile',
              'Amazon',
              'Congo',
              'Mississippi',
              'Rio-Grande'];

rivers.sort((a, b) => {
    console.log(a, b);

    return a.length - b.length;
});


Output:
Amazon Nile
Congo Amazon
Congo Amazon
Congo Nile
Mississippi Congo
Mississippi Amazon
Rio-Grande Amazon
Rio-Grande Mississippi
```

In the above example, the array of rivers is reordered by length and a message is output to the console for each time the `sort()` method invokes the `compare()` function. This experiment proves that the `compare()` function is called multiple times to evaluate and re-evaluate each river included in the array. As a result of this behavior, adding new elements to the array of rivers may be detrimental to the `sort()` method's performance. Although it's impossible to dictate the specific amount of times the `compare()`

function is invoked, we can however decrease the amount of work necessary each time the function is executed, by using the Schwartzian Transform technique.

```
Ex.
const lengths = rivers.map((e, i) => {
    return {index: i, value: e.length};
});
```

Implementation of this technique is accomplished by using the `map()` method to pull the position and length values into a temporary `lengths` array.

```
Ex.
lengths.sort((a, b) => {
    return +(a.value > b.value) ||
           +(a.value === b.value) - 1;
});
```

The elements of the resulting `lengths` array are then sorted and used to build the final output array of rivers ordered by length.

```
Ex.
Input:
const sortedRivers = lengths.map((e) => {
    return rivers[e.index];
});
```

Example continues on next page...

```
console.log(sortedRivers);


Output: ['Nile',
         'Congo',
         'Amazon',
         'Rio-Grande',
         'Mississippi']
```

Author: Trevor Morin