

# Arrays and methods

## Array

Two ways of creating empty array

```
let arr = new Array();  
let arr = [];
```

Create an array with elements in them

```
let arr = ["Apple", "Orange", "Plum"];
```

Array in JavaScript is heterogeneous, you can store element of any type in them.

`arr.length` property to get the number of element in array if you used it properly. Otherwise, if you insert item into an array with a huge gap like below

```
let arr = [];  
arr[999] = 59;  
  
console.log(arr.length); // 1000, it is the index of the last element + 1
```

### `arr.at()`

Normally, if you index an array you can only use positive indexing, i.e. `[0..length - 1]`. You can use negative index with the `arr.at` method.

### pop/push

Use `pop` to remove element from the end of the array

Use `push` to insert element to the end of the array. You can insert multiple items by providing them in the parameter.

Treat array as a stack, first in last out data structure.

### shift/unshift

Use `shift` to dequeue the first element from the head of the array

Use `unshift` to add element to the head of the array.

These two operations are slow takes  $O(n)$  because it requires shifting the array after removing or adding the element.

## Queue

To use an array as a queue, use the `push` and `shift` function to dequeue and enqueue element into the queue.

# Looping over array

You can use a traditional index loop

```
let arr = ["Apple", "Orange", "Pear"];

for (let i = 0; i < arr.length; i++) {
  console.log(arr[i]);
}
```

But if you only need to loop over the element without the indices then there is a shorter syntax

```
let fruits = ["Apple", "Orange", "Plum"];

// iterates over array elements
for (let fruit of fruits) {
  console.log(fruit);
}
```

Remember although you can use `for ... in` loop for array, it is not optimized for looping over array, so it will be slower. In addition, you will be looping over extra properties that is in array. So it is not recommended to use `for ... in` for array iterations, only for object property iterations.

## toString()

Array object implements the `toString()` method to return a comma separated value of String

```
let arr = [1, 2, 3];

arr.toString() // returns "1,2,3"
```

Array object doesn't have `Symbol.toPrimitive` nor a valid `valueOf`, so they only implement `toString` as a catch all object to primitive conversion.

`[]` will become `""`

`[1]` will become `"1"`

`[1, 2]` will become `"1,2"`

## Comparison of array using `==`

Remember that `==` comparison does type conversion if the type are different, and with objects it will be converted to a primitive, and in this case because array only implemented `toString()` it will be converted to a String. Which results in some really interesting comparison.

With `==` if you are comparing two object, it will only be `true` if they are referencing the same object.

The only exception is `null == undefined` is `true` and nothing else.

```
[] == [] // false, because they are different object when you created an empty array
[0] == [0] // false, again different object
```

```
0 == [] // true, because the empty array gets converted to '', and '' is converted to 0 which is true
'0' == [] // false, since empty array converts to '', those two strings are different
```

So how do we actually compare array? Use a loop to compare item-by-item

## Array methods

### `arr.splice`

```
arr.splice(start[, deleteCount, ele1, ..., elemN])
```

The splice method will modify `arr` starting from index `start`, it removes `deleteCount` elements and then inserts `elem1, ..., elemN` at their place. Returns the array of removed elements.

```
let arr = ["Dusk blade", "Eclipse", "Radiant virtue", "Moonstone"];

console.log(arr.splice(1, 2)); // Prints out ["Eclipse", "Radiant virtue"] they are removed
console.log(arr); // Prints out ["Dusk blade", "Moonstone"]
```

## arr.slice

```
arr.slice([start], [end])
```

The slice method will return a new array copying all the item from index `start` to `end` not including `end`.

## arr.concat

```
arr.concat(arg1, arg2...)
```

Create a new array that includes values from other arrays and additional items.

`args` can be either an array, in which it will include every items, or it can be elements themselves which will also be appended into the array

## indexOf, includes

- `arr.indexOf(item, from)`: Looks for `item` starting from index `from`, and return the index where it is found, if it cannot find it return `-1`
- `arr.includes(item, from)`: Looks for `item` starting from index `from`, and return `true` if found

The comparison for `indexOf` and `includes` uses `===` strict equality checking. So if you are looking for let's say `false`, you won't be accidentally finding `0`, since `==` equality checking does type conversion.

## arr.find

If we are looking for an object with specific condition, we can use `arr.find`. This is very similar to `find` in Ruby as well. We will pass in an anonymous function into `find`, and it will find the first element that satisfy the condition, i.e. that returns `true`.

When you are writing the anonymous function, if you don't need all of the arguments, it is okay to write your anonymous function without it. Since if the `find` will pass all three of those parameter into the anonymous function, but if your function doesn't take it it is okay.

```
function foo(a, b) {  
  console.log(a, b);  
}  
  
foo(1, 2, 3); // Calling it with more argument won't matter. Extra arg are ignored  
foo(1); // Calling it with less arg is fine too, rest will just be undefined
```

Example of using `arr.find`:

```
let result = arr.find(function(item, index, array) {  
  // if true is returned, item is returned and iteration is stopped  
  // if no true is return, undefined is returned  
});
```

There are other variant to find `arr.findIndex` if you are interested in finding the index of the element you are looking for instead of the element.

`arr.findLastIndex` to search right to left.

## arr.filter

Very similar syntax to `arr.find`, but instead of searching for only one element, this method will look for all element that satisfies the condition, i.e. that returns `true` and return the result in an array.

## arr.map

Again very similar syntax as well to `find` and `map`, this time you are transforming each element of the array into something else. The function will be responsible for collecting all of the element. You just need to return the element after it is transformed

```
let nums = [1, 2, 3, 4, 5]; // Using multi-line arrow function, to practice  
returning.console.log(nums.map((ele) => { let result = ele * 2; return result;}));
```

## arr.sort

Sorts the array in place. However, by default if you don't provide a comparer function it will be sorting it lexicographically, even if the elements are other type. It will attempt to convert the type into String and then sort it accordingly.

```
let numArr = [1, 2, 15];  
  
numArr.sort(); // This will modify numArr to be [1, 15, 2]! Because lexicographically 2 comes  
after 15!
```

In order to fix this we will have to write our own comparer function.

```
function compareNumeric(a, b) {  
  if (a > b) return 1; // a comes after b  
  if (a == b) return 0; // they are equal return 0  
  return -1; // a comes before b  
}
```

```
let numArr = [1, 2, 15];  
numArr.sort(compareNumeric); // now this will modify numArr to be [1, 2, 15];
```

Actually, you can even simplify this even further by just writing

```
arr.sort((a, b) => a - b); // Ascending  
arr.sort((a, b) => -(a - b)); // Descending
```

## arr.reverse

Just reverse the order of elements in place.

```
let arr = [1, 10, 4, 2];  
arr.reverse(); // arr is now [2, 4, 10, 1];
```

## split and join

Works just like how it work in Python, but with a minor differences.

```
let names = "Ricky Xin Rek'sai"  
  
let arr = names.split(' '); // arr is ["Ricky", "Xin", "Rek'sai"]
```

`split()` need you to specify a space `' '` if your delimiter separating the different words by a space. In addition, if you provide in an empty String `''` as delimiter, it will split the String into array of single letters.

`join()` requires you to call it on the array that you are joining together, and you provide the String to join all the element together by as a parameter

```
let strs = ["Ricky", "Xin", "Rek'sai"];  
  
let out = strs.join("-"); // Becomes "Ricky-Xin-Rek'sai"
```

## reduce/reduceRight

The basic syntax for using `reduce/reduceRight` is as follows:

```
let value = arr.reduce(function(accumulator, item, index, array), initial);
```

- `accumulator`: The result of previous function call, on the first element it is equal to `initial` if it is provided
- `item`: The current array item
- `index`: The current array item's index

- `array`: The array that you call `reduce` on
- `initial`: if the initial is provided then `accumulator` will be assigned that value and start the iteration at the first element. **Otherwise, if `initial` is not provided, then `accumulator` will take the value of the first element of the array and start the iteration at the 2nd element**

How `reduce` work is that the function will be applied to every element, the function will return a result, and that result is passed to the next element as `accumulator`. The first parameter basically becomes the storage area for storing the overall results.

The easiest example is to use this to sum up all the elements in a number array:

```
let nums = [1, 2, 3, 4, 5];

console.log(nums.reduce((sum, ele) => sum + ele), 0); // Will return 15
```

You can do more complicated logic with multi-line arrow function. You just need to return the result as the accumulator for the next function call.

## Array.isArray

You can check if an object is an array by using this function to check whether the parameter passed is an array or not. This is needed because `typeof` doesn't distinguish between `object` and `array`.

---

Revision #4

Created 21 December 2022 20:17:51 by Tamarine

Updated 22 December 2022 03:02:51 by Tamarine