Arrays

Objects allow you to store keyed collections of values. That's fine.

But quite often we find that we need an *ordered collection*, where we have a 1st, a 2nd, a 3rd element and so on. For example, we need that to store a list of something: users, goods, HTML elements etc.

It is not convenient to use an object here, because it provides no methods to manage the order of elements. We can't insert a new property "between" the existing ones. Objects are just not meant for such use.

There exists a special data structure named `Array`, to store ordered collections.

Declaration

There are two syntaxes for creating an empty array:

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

Almost all the time, the second syntax is used. We can supply initial elements in the brackets:

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

Array elements are numbered, starting with zero.

We can get an element by its number in square brackets:

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

alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Plum
```

We can replace an element:

```
fruits[2] = 'Pear'; // now ["Apple", "Orange", "Pear"]
```

...Or add a new one to the array:

```
fruits[3] = 'Lemon'; // now ["Apple", "Orange", "Pear", "Lemon"]
```

The total count of the elements in the array is its `length`:

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

alert( fruits.length ); // 3
```

We can also use `alert` to show the whole array.

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

alert( fruits ); // Apple,Orange,Plum
```

An array can store elements of any type.

For instance:

```
// mix of values
let arr = [ 'Apple', { name: 'John' }, true, function() { alert('hello'); } ];

// get the object at index 1 and then show its name
alert( arr[1].name ); // John

// get the function at index 3 and run it
arr[3](); // hello
```

Trailing comma

An array, just like an object, may end with a comma:

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

The "trailing comma" style makes it easier to insert/remove items, because all lines become alike.

Methods pop/push, shift/unshift

A queue is one of most common uses of an array. In computer science, this means an ordered collection of elements which supports two operations:

- `push` appends an element to the end.
- `shift` get an element from the beginning, advancing the queue, so that the 2nd element becomes the 1st.

Arrays support both operations.

In practice we need it very often. For example, a queue of messages that need to be shown on-screen.

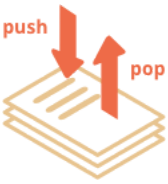
There's another use case for arrays – the data structure named stack.

It supports two operations:

- `push` adds an element to the end.
- `pop` takes an element from the end.

So new elements are added or taken always from the "end".

A stack is usually illustrated as a pack of cards: new cards are added to the top or taken from the top:



For stacks, the latest pushed item is received first, that's also called LIFO (Last-In-First-Out) principle. For queues, we have FIFO (First-In-First-Out).

Arrays in JavaScript can work both as a queue and as a stack. They allow you to add/remove elements both to/from the beginning or the end.

In computer science the data structure that allows it is called deque.

Methods that work with the end of the array:

pop

Extracts the last element of the array and returns it:

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

```

```
alert( fruits.pop() ); // remove "Pear" and alert it

alert( fruits ); // Apple, Orange
```

`push`

Append the element to the end of the array:

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

fruits.push("Pear");

alert( fruits ); // Apple, Orange, Pear
```

The call `fruits.push(...)` is equal to `fruits[fruits.length] = ...`.

Methods that work with the beginning of the array:

`shift`

Extracts the first element of the array and returns it:

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

alert( fruits.shift() ); // remove Apple and alert it

alert( fruits ); // Orange, Pear
```

`unshift`

Add the element to the beginning of the array:

```
let fruits = ["Orange", "Pear"];

fruits.unshift('Apple');

alert( fruits ); // Apple, Orange, Pear
```

Methods `push` and `unshift` can add multiple elements at once:

```
let fruits = ["Apple"];

fruits.push("Orange", "Peach");
fruits.unshift("Pineapple", "Lemon");

// ["Pineapple", "Lemon", "Apple", "Orange", "Peach"]
alert( fruits );
```

Internals

An array is a special kind of object. The square brackets used to access a property `arr[0]` actually come from the object syntax. Numbers are used as keys.

They extend objects providing special methods to work with ordered collections of data and also the `length` property. But at the core it's still an object.

Remember, there are only 7 basic types in JavaScript. Array is an object and thus behaves like an object.

For instance, it is copied by reference:

```
let fruits = ["Banana"]

let arr = fruits; // copy by reference (two variables reference the same array)

alert( arr === fruits ); // true

arr.push("Pear"); // modify the array by reference

alert( fruits ); // Banana, Pear - 2 items now
```

...But what makes arrays really special is their internal representation. The engine tries to store its elements in the contiguous memory area, one after another, just as depicted on the illustrations in this chapter, and there are other optimizations as well, to

make arrays work really fast.

But they all break if we quit working with an array as with an "ordered collection" and start working with it as if it were a regular object.

For instance, technically we can do this:

```
let fruits = []; // make an array

fruits[99999] = 5; // assign a property with the index far greater than its length

fruits.age = 25; // create a property with an arbitrary name
```

That's possible, because arrays are objects at their base. We can add any properties to them.

But the engine will see that we're working with the array as with a regular object. Array-specific optimizations are not suited for such cases and will be turned off, their benefits disappear.

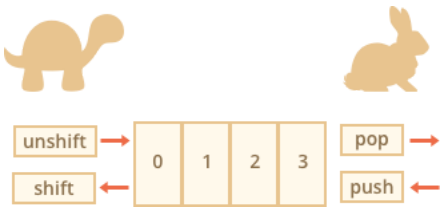
The ways to misuse an array:

- Add a non-numeric property like `arr.test = 5`.
- Make holes, like: add `arr[0]` and then `arr[1000]` (and nothing between them).
- Fill the array in the reverse order, like `arr[1000]`, `arr[999]` and so on.

Please think of arrays as special structures to work with the *ordered data*. They provide special methods for that. Arrays are carefully tuned inside JavaScript engines to work with contiguous ordered data, please use them this way. And if you need arbitrary keys, chances are high that you actually require a regular object `{}`.

Performance

Methods `push/pop` run fast, while `shift/unshift` are slow.



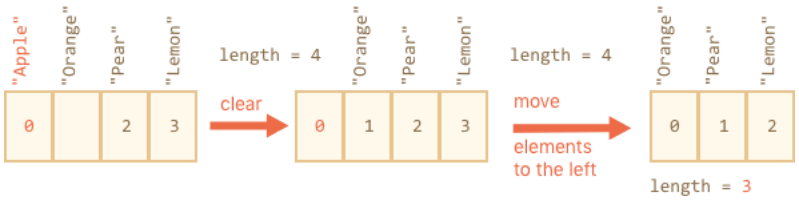
Why is it faster to work with the end of an array than with its beginning? Let's see what happens during the execution:

```
fruits.shift(); // take 1 element from the start
```

It's not enough to take and remove the element with the number `0`. Other elements need to be renumbered as well.

The `shift` operation must do 3 things:

1. Remove the element with the index `0`.
2. Move all elements to the left, renumber them from the index `1` to `0`, from `2` to `1` and so on.
3. Update the `length` property.



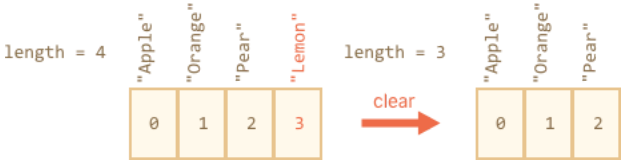
The more elements in the array, the more time to move them, more in-memory operations.

The similar thing happens with `unshift`: to add an element to the beginning of the array, we need first to move existing elements to the right, increasing their indexes.

And what's with `push/pop`? They do not need to move anything. To extract an element from the end, the `pop` method cleans the index and shortens `length`.

The actions for the `pop` operation:

```
fruits.pop(); // take 1 element from the end
```



The `pop` method does not need to move anything, because other elements keep their indexes. That's why it's blazingly fast.

The similar thing with the `push` method.

Loops

One of the oldest ways to cycle array items is the `for` loop over indexes:

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

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

But for arrays there is another form of loop, `for..of`:

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

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

The `for..of` doesn't give access to the number of the current element, just its value, but in most cases that's enough. And it's shorter.

Technically, because arrays are objects, it is also possible to use `for..in`:

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

for (let key in arr) {
  alert( arr[key] ); // Apple, Orange, Pear
}
```

But that's actually a bad idea. There are potential problems with it:

1. The loop `for..in` iterates over *all properties*, not only the numeric ones.

   There are so-called "array-like" objects in the browser and in other environments, that *look like arrays*. That is, they have `length` and indexes properties, but they may also have other non-numeric properties and methods, which we usually don't need. The `for..in` loop will list them though. So if we need to work with array-like objects, then these "extra" properties can become a problem.

2. The `for..in` loop is optimized for generic objects, not arrays, and thus is 10-100 times slower. Of course, it's still very fast. The speedup may only matter in bottlenecks or seem irrelevant. But still we should be aware of the difference.

Generally, we shouldn't use `for..in` for arrays.

A word about "length"

The `length` property automatically updates when we modify the array. To be precise, it is actually not the count of values in the array, but the greatest numeric index plus one.

For instance, a single element with a large index gives a big length:

```
let fruits = [];
fruits[123] = "Apple";

alert( fruits.length ); // 124
```

Note that we usually don't use arrays like that.

Another interesting thing about the `length` property is that it's writable.

If we increase it manually, nothing interesting happens. But if we decrease it, the array is truncated. The process is irreversible, here's the example:

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

arr.length = 2; // truncate to 2 elements
alert( arr ); // [1, 2]

arr.length = 5; // return length back
alert( arr[3] ); // undefined: the values do not return
```

So, the simplest way to clear the array is: `arr.length = 0;` .

new Array()

There is one more syntax to create an array:

```
let arr = new Array("Apple", "Pear", "etc");
```

It's rarely used, because square brackets `[]` are shorter. Also there's a tricky feature with it.

If `new Array` is called with a single argument which is a number, then it creates an array *without items, but with the given length*.

Let's see how one can shoot themself in the foot:

```
let arr = new Array(2); // will it create an array of [2] ?

alert( arr[0] ); // undefined! no elements.

alert( arr.length ); // length 2
```

In the code above, `new Array(number)` has all elements `undefined` .

To evade such surprises, we usually use square brackets, unless we really know what we're doing.

Multidimensional arrays

Arrays can have items that are also arrays. We can use it for multidimensional arrays, to store matrices:

```
let matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

alert( matrix[1][1] ); // the central element
```

toString

Arrays have their own implementation of `toString` method that returns a comma-separated list of elements.

For instance:

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

alert( arr ); // 1,2,3
alert( String(arr) === '1,2,3' ); // true
```

Also, let's try this:

```
alert( [] + 1 ); // "1"
alert( [1] + 1 ); // "11"
alert( [1,2] + 1 ); // "1,21"
```

Arrays do not have `Symbol.toPrimitive` , neither a viable `valueOf` , they implement only `toString` conversion, so here `[]` becomes an empty string, `[1]` becomes `"1"` and `[1,2]` becomes `"1,2"` .

When the binary plus `"+"` operator adds something to a string, it converts it to a string as well, so the next step looks like this:

```
alert( "" + 1 ); // "1"
alert( "1" + 1 ); // "11"
alert( "1,2" + 1 ); // "1,21"
```

Summary

Array is a special kind of object, suited to storing and managing ordered data items.

- The declaration:

```
// square brackets (usual)
let arr = [item1, item2...];

// new Array (exceptionally rare)
let arr = new Array(item1, item2...);
```

  The call to `new Array(number)` creates an array with the given length, but without elements.

- The `length` property is the array length or, to be precise, its last numeric index plus one. It is auto-adjusted by array methods.
- If we shorten `length` manually, the array is truncated.

We can use an array as a deque with the following operations:

- `push(...items)` adds `items` to the end.
- `pop()` removes the element from the end and returns it.
- `shift()` removes the element from the beginning and returns it.
- `unshift(...items)` adds items to the beginning.

To loop over the elements of the array:

- `for (let i=0; i<arr.length; i++)` – works fastest, old-browser-compatible.
- `for (let item of arr)` – the modern syntax for items only,
- `for (let i in arr)` – never use.

We will return to arrays and study more methods to add, remove, extract elements and sort arrays in the chapter Array methods.

Tasks

Is array copied?

importance: 3

What is this code going to show?

```
let fruits = ["Apples", "Pear", "Orange"];

// push a new value into the "copy"
let shoppingCart = fruits;
shoppingCart.push("Banana");

// what's in fruits?
alert( fruits.length ); // ?
```

solution

The result is `4`:

```
let fruits = ["Apples", "Pear", "Orange"];

let shoppingCart = fruits;
```

```
4
5 shoppingCart.push("Banana");
6
7 alert( fruits.length ); // 4
```

That's because arrays are objects. So both `shoppingCart` and `fruits` are the references to the same array.

Array operations.

importance: 5

Let's try 5 array operations.

1. Create an array `styles` with items "Jazz" and "Blues".
2. Append "Rock-n-Roll" to the end.
3. Replace the value in the middle by "Classics". Your code for finding the middle value should work for any arrays with odd length.
4. Strip off the first value of the array and show it.
5. Prepend `Rap` and `Reggae` to the array.

The array in the process:

```
1 Jazz, Blues
2 Jazz, Bues, Rock-n-Roll
3 Jazz, Classics, Rock-n-Roll
4 Classics, Rock-n-Roll
5 Rap, Reggae, Classics, Rock-n-Roll
```

solution

```
1 let styles = ["Jazz", "Blues"];
2 styles.push("Rock-n-Roll");
3 styles[Math.floor((styles.length - 1) / 2)] = "Classics";
4 alert( styles.shift() );
5 styles.unshift("Rap", "Reggae");
```

Calling in an array context

importance: 5

What is the result? Why?

```
1 let arr = ["a", "b"];
2
3 arr.push(function() {
4   alert( this );
5 })
6
7 arr[2](); // ?
```

solution

The call `arr[2]()` is syntactically the good old `obj[method]()`, in the role of `obj` we have `arr`, and in the role of `method` we have `2`.

So we have a call of the function `arr[2]` as an object method. Naturally, it receives `this` referencing the object `arr` and outputs the array:

```
1 let arr = ["a", "b"];
2
```

```
arr.push(function() {
  alert( this );
})

arr[2](); // "a","b",function
```

The array has 3 values: initially it had two, plus the function.

Sum input numbers

importance: 4

Write the function `sumInput()` that:

- Asks the user for values using `prompt` and stores the values in the array.
- Finishes asking when the user enters a non-numeric value, an empty string, or presses "Cancel".
- Calculates and returns the sum of array items.

P.S. A zero `0` is a valid number, please don't stop the input on zero.

Run the demo solution

Please note the subtle, but important detail of the solution. We don't convert `value` to number instantly after `prompt`, because after `value = +value` we would not be able to tell an empty string (stop sign) from the zero (valid number). We do it later instead.

```
function sumInput() {

  let numbers = [];

  while (true) {

    let value = prompt("A number please?", 0);

    // should we cancel?
    if (value === "" || value === null || !isFinite(value)) break;

    numbers.push(+value);
  }

  let sum = 0;
  for (let number of numbers) {
    sum += number;
  }
  return sum;
}

alert( sumInput() );
```

A maximal subarray

importance: 2

The input is an array of numbers, e.g. `arr = [1, -2, 3, 4, -9, 6]`.

The task is: find the contiguous subarray of `arr` with the maximal sum of items.

Write the function `getMaxSubSum(arr)` that will return that sum.

For instance:

```
getMaxSubSum([-1, 2, 3, -9]) = 5 (the sum of highlighted items)
getMaxSubSum([2, -1, 2, 3, -9]) = 6
getMaxSubSum([-1, 2, 3, -9, 11]) = 11
getMaxSubSum([-2, -1, 1, 2]) = 3
getMaxSubSum([100, -9, 2, -3, 5]) = 100
getMaxSubSum([1, 2, 3]) = 6 (take all)
```

If all items are negative, it means that we take none (the subarray is empty), so the sum is zero:

```
getMaxSubSum([-1, -2, -3]) = 0
```

Please try to think of a fast solution: $O(n^2)$ or even O(n) if you can.

Open a sandbox with tests.

solution

The slow solution

Fast solution

Previous lesson Next lesson

Share

Tutorial map

Comments

- You're welcome to post additions, questions to the articles and answers to them.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen...)
- If you can't understand something in the article – please elaborate.