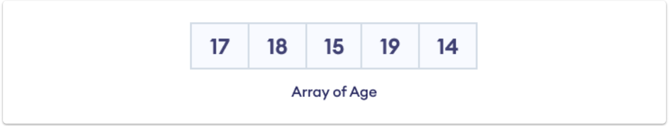
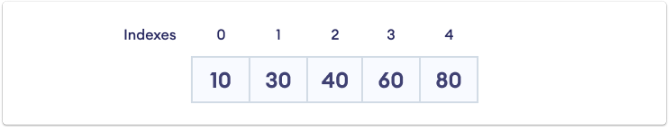
12-dars. JS massivlar va massiv metodlari (for, for...of, for…in, forEach, map va filter,every va some, reduce va reduceRight, find, sort, …)

12-dars. JS massivlar va massiv metodlari (for, for...of, for…in, forEach, map va filter,every va some, reduce va reduceRight, find, sort, …)

Arrays

> *Suppose we need to record the age of 5 students. Instead of creating 5 separate variables, we can simply create an array*





Creating Arrays

- Array literals
- The ...spread operator on an iterable object
- The Array() constructor
- The Array.of() and Array.from() factory methods

Array Literals

> Note
>
> By far the simplest way to create an array is with an array literal, which is simply a comma-separated list of array elements within square brackets

```
let empty = []; // An array with no elements
let primes = [2, 3, 5, 7, 11]; // An array with 5 numeric elements
let misc = [ 1.1, true, "a", ]; // 3 elements of various types + trailing comma
```

```
let base = 1024;
let table = [base, base+1, base+2, base+3];
```

```
let b = [[1, {x: 1, y: 2}], [2, {x: 3, y: 4}]];
```

```
let count = [1,,3]; // Elements at indexes 0 and 2. No element at index 1
let undefs = [,,]; // An array with no elements but a length of 2
```

The ...spread operator on an iterable object

> Note
>
> In ES6 and later, you can use the “spread operator,” ..., to include the elements of one array within an array literal:

```
let a = [1, 2, 3];
let b = [0,  ... a, 4]; // b == [0, 1, 2, 3, 4]
```

```
let original = [1,2,3];
let copy = [ ... original];
copy[0] = 0; // Modifying the copy does not change the original
original[0]
```

```
let digits = [ ... "0123456789ABCDEF"];
digits // ⇒["0","1","2","3","4","5","6","7","8","9","A","B","C","D","E", "F"]
```

The Array() constructor

> *Create an array is with the Array() constructor. You can invoke this constructor in three distinct ways:*

1. Call it with no arguments:

```
let a = new Array();
```

> *This method creates an empty array with no elements and is equivalent to the array literal [].*

2. Call it with a single numeric argument, which specifies a length:

```
let a = new Array(10);
a.length //=> 10
```

3. Explicitly specify two or more array elements or a single nonnumeric element for the array:

```
let a = new Array(5, 4, 3, 2, 1, "testing, testing", true, false, {name:"john"});
```

> *[] > Array()*

The Array.of() and Array.from() factory methods

`Array.of()`

> *When the Array() constructor function is invoked with one numeric argument, it uses that argument as an array length. But when invoked with more than one numeric argument, it treats those arguments as elements for the array to be created. This means that the Array() constructor cannot be used to create an array with a single numeric element.*

> *In ES6, the Array.of() function addresses this problem: it is a factory method that creates and returns a new array, using its argument values (regardless of how many of them there are) as the array elements:*

`Array constructor 1 bitta qiymat berilsa o'sha qiymatga teng uzunlikdagi bo'sh Array yaratidi!.`

`Array.of() bo'lsa u faqat argument qabul qiladi va o'shani qaytardi!`

```
Array.of() // ⇒ []; returns empty array with no arguments
Array.of(10) // ⇒ [10]; can create arrays with a single numeric argument
Array.of(1,2,3) // ⇒ [1, 2, 3]
```

`Array.from()`

> *Array.from is another array factory method introduced in ES6. It expects an iterable or array-like object as its first argument and returns a new array that contains the elements of that object. With an iterable argument, Array.from(iterable) works like the spread operator [...iterable] does. It is also a simple way to make a copy of an array:*

```
let copy = Array.from(original);
```

```
let truearray = Array.from(arraylike);
```

Reading and Writing Array Elements

Methods pop/push, shift/unshift

> *You access an element of an array using the [] operator.*

```
let a = ["world"]; // Start with a one-element array
let value = a[0]; // Read element 0
a[1] = 3.14; // Write element 1
let i = 2;
a[i] = 3; // Write element 2
a[i + 1] = "hello"; // Write element 3
a[a[i]] = a[0]; // Read elements 0 and 2, write element 3
a.length // ⇒
```

```
a[-1.23] = true; // This creates a property named "-1.23"
a["1000"] = 0; // This the 1001st element of the array
a[1.000] = 1; // Array index 1. Same as a[1] = 1;
```

```
let a = [true, false]; // This array has elements at indexes 0 and 1 32
a[2] // ⇒ undefined; no element at this index.
a[-1] // ⇒ undefined; no property with this name.
```

Adding and Deleting Array Elements

> *Just assign values to new indexes:*

```
let a = []; // Start with an empty array.
a[0] = "zero"; // And add elements to it. a[1] = "one";
```

> *You can delete array elements with the delete operator, just as you can delete object properties:*

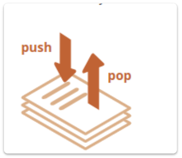
```
let a = [1,2];
delete a[2]; // a now has no element at index 2
2 in a // ⇒ false: no array index 2 is defined
a.length
```

Methods pop/push, shift/unshift

- `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.

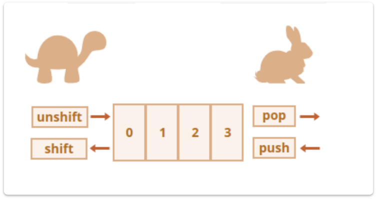
Arrays support both operations.

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



[Performance]

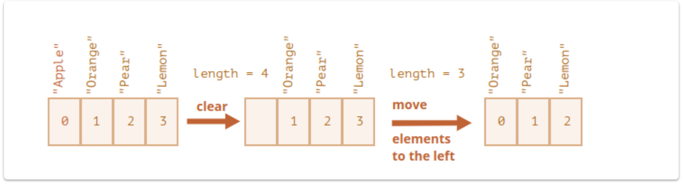
> *Methods `push/pop` run fast, while `shift/unshift` are slow.*



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

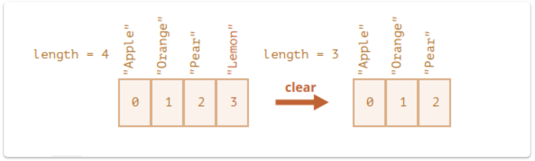
The `shift` operation must do 3 things:

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



Array Length

> *The length property specifies the number of elements in the array.*

```
[].length // ⇒ 0: the array has no elements
["a","b","c"].length // ⇒ 3: highest index is 2, length is 3
```

```
a = [1,2,3,4,5]; // Start with a 5-element array.
a.length = 3; // a is now [1,2,3].
a.length = 0; // Delete all elements. a is [].
a.length = 5; // Length is 5, but no elements, like new Array(5)
```

Array Methods

- Iterator methods loop over the elements of an array, typically invoking a function that you specify on each of those elements.
- Stack and queue methods add and remove array elements to and from the beginning and the end of an array.
- Sub array methods are for extracting, deleting, inserting, filling, and copying contiguous regions of a larger array.
- Searching and sorting methods are for locating elements within an array and for sorting the elements of an array.

Array Iterator Methods

> *The methods described in this section iterate over arrays by passing array elements, in order, to a function you supply, and they provide convenient ways to iterate, map, filter, test, and reduce arrays*

FOREACH()

> *The forEach() method iterates through an array, invoking a function you specify for each element.*

```
let data = [1,2,3,4,5], sum = 0; // Compute the sum of the elements of the array
data.forEach(value ⇒ {
        sum += value;
}); // sum == 15 // Now increment each array element
data.forEach(function(v, i, a) {
        a[i] = v + 1;
}); // data == [2,3,4,5,6]
```

> **Note**
>
> Note that forEach() does not provide a way to terminate iteration before all elements have been passed to the function. That is, there is no equivalent of the break statement you can use with a regular for loop.

MAP()

> *The map() method passes each element of the array on which it is invoked to the function you specify and returns an array containing the values returned by your function*

```
let a = [1, 2, 3];
a.map(x ⇒ x*x) // ⇒ [1, 4, 9]: the function takes input x and returns x*x
```

> **Note**
>
> Note that map() returns a new array: it does not modify the array it is invoked on

FILTER()

> *The function you pass to it should be predicate: a function that returns true or false.*

```
let a = [5, 4, 3, 2, 1];
a.filter(x => x < 3) // => [2, 1]; values less than 3
a.filter((x,i) => i%2 === 0) // => [5, 3, 1]; every other value
```

```
a = a.filter(x => x !== undefined && x !== null);
```

FILL()

> *The fill() method sets the elements of an array, or a slice of an array, to a specified value*

```
let a = new Array(5); // Start with no elements and length 5
a.fill(0) // => [0,0,0,0,0]; fill the array with zeros
a.fill(9, 1) // => [0,9,9,9,9]; fill with 9 starting at index 1
a.fill(8, 2, -1) // => [0,9,8,8,9]; fill with 8 at indexes 2, 3
```

INDEXOF() AND LASTINDEXOF()

> *indexOf() and lastIndexOf() search an array for an element with a specified value and return the index of the first such element found, or -1 if none is found. indexOf() searches the array from beginning to end, and lastIndexOf() searches from end to beginning:*

```
let a = [0,1,2,1,0];
a.indexOf(1) // => 1: a[1] is 1
a.lastIndexOf(1) // => 3: a[3] is 1
a.indexOf(3)
```

INCLUDES()

> *The ES2016 includes() method takes a single argument and returns true if the array contains that value or false otherwise. It does not tell you the index of the value, only whether it exists*

```
let a = [1,true,3,NaN];
a.includes(true) // => true
a.includes(2) // => false
a.includes(NaN) // => true
a.indexOf(NaN) // => -1; indexOf can't find NaN
```

SORT()

> *sort() sorts the elements of an array in place and returns the sorted array.*

```
let a = ["banana", "cherry", "apple"];
a.sort(); // a == ["apple", "banana", "cherry"]
```

```
let a = [33, 4, 1111, 222];
a.sort(); // a == [1111, 222, 33, 4]; alphabetical order
a.sort(function(a,b) { // Pass a comparator function
        return a-b; // Returns < 0, 0, or > 0, depending on order
}); // a == [4, 33, 222, 1111]; numerical order
a.sort((a,b) => b-a); // a == [1111, 222, 33, 4]; reverse numerical order
```

[Summary]

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.

Getting the elements:

- we can get element by its index, like `arr[0]`
- also we can use `at(i)` method that allows negative indexes. For negative values of `i`, it steps back from the end of the array. If `i >= 0`, it works same as `arr[i]`.

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.

To compare arrays, don't use the `==` operator (as well as `>`, `<` and others), as they have no special treatment for arrays. They handle them as any objects, and it's not what we usually want.

Instead you can use `for..of` loop to compare arrays item-by-item.

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

```
Dataview (inline field '='): Error:
-- PARSING FAILED --------------------------------------------------

> 1 | =
    | ^

Expected one of the following:

'(', 'null', boolean, date, duration, file link, list ('[1, 2, 3]'), negated field, number, object ('{ a: 1, b: 2 }'), string, variable
```