



# JavaScript Arrays

Asst.Prof. Dr. Umaporn Supasitthimethee

ผศ.ดร.อุมาพร สุภสีทธิเมธี

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)

JavaScript: The Definitive Guide, Seventh Edition, by David Flanagan

---



# Arrays

- An array is **an ordered collection of values**. **JavaScript arrays are object**.
- **Each value is called an element**, and each element has a numeric position in the array, known as its index (***zero-based index***).
- JavaScript **arrays are untyped**: an array element may be of **any type**, and different elements of the **same array may be of different types**.
- **Array elements** may even **be objects or other arrays**, which allows you to create complex data structures such as arrays of objects and arrays of arrays.
- **JavaScript arrays are dynamic**: they **grow or shrink as needed**, and there is no need to declare a fixed size for the array when you create it or to reallocate it when the size changes.
- Every **JavaScript array** has a **length property**.



# Creating Arrays

//05\_Arrays/script2.js

1. Array literals
2. The ... spread operator on an iterable object
3. The Array() constructor
4. The Array.of() and Array.from() factory methods

# 1. Array literals

- 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

//05\_Arrays/script1.js

```
let arr1 = [10, 'in progress', true];

let arr2=  [15 ,30 ,42];

let students = [
  { id: 1, name: 'Ann' },
  { id: 2, name: 'Peter' },
  { id: 3, name: 'Mary' }
];

let colors = [
  ['pink', 'red'],
  ['yellow', 'orange', 'brown']
];
```



## 2. The ... spread operator on an iterable object

- 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]
```

- The three dots “spread” the array so that its elements become elements within the array literal that is being created.


## 2. The ... spread operator on an iterable object

- The spread operator is a convenient way to create a (shallow) copy of an array:

//05\_Arrays/script2.js

```
let c = [5, 10, 15];  
let d = [...c];  
d[0] = 10;  
console.log(`d: ${d}`); //d: 10,10,15  
console.log(`c[0]: ${c[0]}`); //5  
console.log(`d[0]: ${d[0]}`); //10
```

- Modifying the copy does not change the original



## 3. The `Array()` Constructor

- Call it with **no arguments**:

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

- Call it with a single numeric argument, which **specifies a length**:

```
let a = new Array(10);
```

- Explicitly **specify two or more array elements** or a **single non-numeric element for the array**:

```
let a = new Array(3, 2, 1, "testing");
```

## 4. The `Array.of()` and `Array.from()` factory methods

- 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:

`//05_Arrays/script2.js`

```
Array.of()           // => []; returns empty array with no arguments
Array.of(5)          // => [5]; create arrays with a single numeric argument
Array.of(1,2,3)       // => [1, 2, 3]
```



## 4. The `Array.of()` and `Array.from()` factory methods

- `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:

`//05_Arrays/script2.js`

```
let j = Array.of(1, 2, 3);  
  
let k = Array.from(j); //k: 1,2,3
```



# Reading and Writing Array Elements

- You access an element of an array using the `[]` operator.
- An arbitrary expression that has a non-negative integer value should be inside the brackets.
- You can use this syntax to both read and write the value of an element of an array. Thus, the following are all legal JavaScript statements:

```
let a = ["hello"];  
let value = a[0];           // Read element 0  
a[1] = 3.5;                 // Write element 1  
let i = 2;  
a[i] = 3;                   // Write element 2  
a[i + 1] = "world";         // Write element 3  
a[a[i]] = a[0];             // Read elements 0 and 3, write element 3
```

# Adding and Deleting Array Elements

//05\_Arrays/script4.js

```
let arrList = []; // Start with an empty array.  
arrList[0] = 10; // add elements to it.  
arrList[1] = 20; // add elements to it.  
arrList[2] = 'ten'; // add elements to it.  
delete arrList[1]; // delete element at index 1  
arrList.length //length=3
```

```
//result  
[ 10, <1 empty item>, 'ten' ]
```

**Note that** using delete on an array element does **not alter the length property** and **does not shift elements with higher indexes down to fill in the gap** that is left by the deleted property.

# Iterating Arrays

- As of ES6, the easiest way to loop through each of the elements of an array (or any iterable object) is with the for/of loop

//05\_Arrays/script3.js

```
let letters = [...'Hello world']; //spread array of characters
let msg = '';
for (let ch of letters) {
  msg += ch + ', ';
}
console.log(msg);
```

//result

```
H, e, l, l, o, , w, o, r, l, d,
```

# destructuring assignment

- The **destructuring assignment** syntax is a JavaScript expression that makes it possible to **unpack values from arrays, or properties from objects, into distinct variables.**

//05\_Arrays/script5.js

```
let a, b, rest;  
[a, b] = [5, 10];  
  
console.log(a); //5  
console.log(b); //10  
  
[a, b, ...rest] = [5, 10, 15, 20, 25];  
console.log(rest); // [15,20,25]
```

```
({ a, b } = { a: 10, b: 20 });  
console.log(a); // 10  
console.log(b); // 20  
  
({ a, b, ...rest } = { a: 10, b: 20, c: 30, d: 40 });  
console.log(a); // 10  
console.log(b); // 20  
console.log(rest); // {c: 30, d: 40}
```

# Iterating Arrays (with index of each array element)

- If you want to use a **for/of loop** for an array and **need to know the index** of each array element, use the `entries()` method of the array, along with destructuring assignment.

`//05_Arrays/script3.js`

```
let letters = [...'Hello world'];
let value = '';
for (let [index, letter] of letters.entries()) {
  if (index % 2 === 0) value += letter; // letters at even indexes
}
console.log(`value: ${value}`); // "Hlowrd"
```

The **entries()** method returns a new **Array Iterator** object that contains the key/value pairs for each index in the array



# Array Methods

**Array Iterator Methods:** Iterator methods loop over the elements of an array

- **forEach()** iterates through an array, invoking a function you specify for each element
- **map()** 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.
- **filter()** returns an array containing a subset of the elements of the array on which it is invoked.
- **find()** returns the matching element, If no matching element is found, find() returns undefined
- **findIndex()** returns the index of the matching element. If no matching element is found, find() returns -1
- **every()** and **some()** they apply a predicate function you specify to the elements of the array, then return true or false.
- **reduce()** combine the elements of an array, using the function you specify, to produce a single value.



# Array Methods

**Stack and queue methods** add and remove array elements to and from the beginning and the end of an array.

- **push()** appends one or more new elements to the end of an array and returns the new length of the array.  
**pop()** deletes the last element of an array, decrements the array length, and returns the value that it removed.
- **unshift()** adds an element or elements to the beginning of the array, shifts the existing array elements up to higher indexes to make room, and returns the new length of the array.
- **shift()** removes and returns the first element of the array, shifting all subsequent elements down one place to occupy the newly vacant space at the start of the array.





# Array Methods

**Subarray methods** are for extracting, deleting, inserting, filling, and copying contiguous regions of a larger array.

- **slice()** returns a slice, or subarray, of the specified array. Its two arguments specify the start and end of the slice to be returned.
- **splice()** a general-purpose method for inserting or removing elements from an array.
- **fill()** sets the elements of an array, or a slice of an array, to a specified value. It mutates the array it is called on, and also returns the modified array:



# Array Methods

**Searching and sorting methods** are for locating elements within an array and for sorting the elements of an array.

- **indexOf()** 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.
- **includes()** 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.
- **sort()** sorts the elements of an array in place and returns the sorted array.
- **reverse()** reverses the order of the elements of an array and returns the reversed array.



# Array Methods

## Array to String Conversions

- **join()** converts all the elements of an array to strings and concatenates them, returning the resulting string.



# Function Expressions

## Function expression

```
const getRectangleArea = function(width, height) {  
    return width * height;  
};
```

## Named function expression

```
let fact = function factorial(n) {  
    console.log(n);  
    if (n <= 1) {  
        return 1;  
    }  
    return n * factorial(n - 1);  
};  
  
fact (5); //120
```

# Arrow Function Expressions

**Arrow Function expression** is a compact alternative to a traditional function expression but is limited and can't be used in all situations.

```
// Traditional Function (no arguments)
let a = 4;
let b = 2;
function (){
    return a + b + 100;
}

// Arrow Function
let a = 4;
let b = 2;
() => a + b + 100;
```

```
// No param
() => expression

// One param
param => expression

// Multiple param
(param1, paramN) => expression

// Multiline statements
param1 => {
    statement1;
    ...
    statementN;
}

(param1, paramN) => {
    statement1;
    ...
    statementN;
}
```

## Comparing traditional functions to arrow functions

```
// Traditional Function (one argument)
function (a){
    return a + 100;
}

// Arrow Function Break Down
// 1. Remove the word "function" and place arrow
// between the argument and opening body bracket
(a)=> {
    return a + 100;
}

// 2. Remove the body braces and word "return" --
// the return is implied.
(a)=> a + 100;

// 3. Remove the argument parentheses
a => a + 100;
```

```
// Traditional Function (multiple arguments)
function (a, b){
    return a + b + 100;
}

// Arrow Function
(a, b) => a + b + 100;
```

```
// Traditional Function (multiline statements)
function (a, b){
    let chuck=42;
    return a + b + chuck;
}

// Arrow Function
(a, b) => {
    let chuck=42;
    a + b + chuck;
}
```