JS Basic Arrays and Functions

Asst.Prof. Dr. Umaporn Supasitthimethee
ผศ.ดร.อุมาพร สุภสิทธิเมธี

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions

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

Basic Arrays

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.

Array Array array

- **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**.

value element

Creating Arrays

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 [ ]

```
let nums = [15, 30, 42]

let diffArr = [10, 'in progress', true]

let students = [ nested with object  object  array
  { id: 1, name: 'Ann'   },
  { id: 2, name: 'Peter' },
  { id: 3, name: 'Mary' }
]

let colors = [ nested array  array  array
  ['yellow', 'red', 'orange'],
  ['blue', 'green', 'purple']
]
```

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]
```

```
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
```

3. The `Array()` Constructor

java

- 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()` 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:

```
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.from()` factory methods

spread operator

- `Array.from` is another array factory method introduced in ES6. It 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 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[2] = 3  a[i] = 3              // Write element 2
a[2+1] = "world"   a[i + 1] = "world"    // Write element 3
    a[a[2]] = a[0] a[a[i]] = a[0] a[0] = "world"// Read elements 0 and 3, write element 3
```

Adding and Deleting Array Elements

```
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.

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.**

```
// array destructuring
const [a, b] = [5, 10]
console.log(a) // 5
console.log(b) // 10

const [m] = [10, 20, 30, 40]
console.log(m) // 10

const [, , n] = [8, 16, 24, 32]
console.log(n) // 24

//Rest Operator Works in a Destructuring Assignment
const [x, y, ...z] = [5, 10, 15, 20, 25] //with rest operator
console.log(z) // [15,20,25]
```

rest = () destucturingspread = () copy array

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

```
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,
```

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.

```
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

Basic Functions

Functions

- A function is a **block of JavaScript code that is defined once** but may be **executed**, or invoked, **any number of times**.
- **JavaScript functions are parameterized:** a function definition may include a list of identifiers, known as parameters, that work as local variables for the body of the function.
- typprof "function"

  In JavaScript, **functions are objects**, and they can be manipulated by programs. JavaScript can **assign functions to variables and pass them to other functions**

Higher-Order Functions

A “higher-order function” is a function that accepts functions as parameters and/or returns a function.

- JavaScript Functions are **first-class citizens**
  - be assigned to variables (and treated as a value)
  - be passed as an argument of another function
  - be returned function as a value from another function

```
//1. store functions in variables

function add(n1, n2) {
  return n1 + n2
}
let sum = add

let addResult1 = add(10, 20)
let addResult2 = sum(10, 20)

console.log(`add result1: ${addResult1}`)
console.log(`add result2: ${addResult2}`)
```

```
//2. Passing a function to another function
function operator(n1, n2, fn) {
  return fn(n1, n2)
}
function multiply(n1, n2) {
  return n1 * n2
}

let addResult3 = operator(5, 3, add)
let multiplyResult = operator(5, 3, multiply)

console.log(`add result3 : ${addResult3}`)
console.log(`multiply result: ${multiplyResult}`)
```

```
//3. return function as value of another function
function sayGoodBye(){
    return 'Good bye'
}
function doSomething(){
    return sayGoodBye
}
let doIt=doSomething()
console.log(doIt())
```

Function Declarations

Function declaration:

- the function keyword
- the name of the function
- a list of parameters to the function
- the JavaScript statements that define the function, enclosed in curly brackets, {...}.

```
function name([param1[, param2[, ..., paramN]]]) {
     statements

}
```

Function Expressions

Function expressions look a lot like function declarations, but they appear within the context of a larger expression or statement, and *the name is optional*. However, a name can be provided with a function expression.

Function expression: function expression defines a function and assign it to a variable

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

Named function expression : Function expressions can include names, which is useful for recursion.

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

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function

Calling Functions

- *Defining* a function does not *execute* it. Defining it names the function and specifies what to do when the function is called.
- Calling the function actually performs the specified actions with the indicated parameters.

```
//function declaration
function square (side) {
         return side * side
}

square(3); //calling functions
```

```
//function expression
let area=function square(side){
         return side* side
}
area(3); //calling functions
```

Functions must be *in scope* when they are called, but the function declaration can be hoisted:

```
square(3); //hoisting

function square (side) {
         return side * side
}
```

function hoisting only works with function *declarations* — not with function *expressions*.

Primitive Parameter Passing

- **Primitive parameters** are passed to functions **by value**; the value is passed to the function, but if the function changes the value of the parameter, **this change is not reflected globally or in the calling function**.

```
function square(side) {
  return side * side
}
let theSide = 2
console.log(square(theSide)) //4
console.log(theSide) //2
```

Object Parameter Passing

- **Object parameter** (i.e., a non-primitive value, such as Array or a user-defined object) are passed to function and the function changes the object's properties, **that change is visible** outside the function.

```
function myFunc(theObject) {
  theObject.model = "A9999"
}
const product = {model: "A1001", price: 199}
console.log(product.model) // "A1001"

myFunc(product);
console.log(product.model) // "A9999"
```