JavaScript - Arrays

Adapted from https://www.w3schools.com/js/js_arrays.asp

Arrays in JavaScript

```
const array_name = [item1, item2, ...];
```

- Example:

```
const cars = ["Saab", "Volvo", "BMW"];
```

Declaration on many lines

```
const cars = [
  "Saab",
  "Volvo",
  "BMW"
];
```

Using the *new* keyword

```
const cars
= new Array("Saab", "Volvo", "BMW");
```

Another way

```
const cars = [];

cars[0]= "Saab";
cars[1]= "Volvo";
cars[2]= "BMW";
```

- Array index starts at 0

Another other way

```
const points = new Array();
```

- Creates an empty array named points

Mixed types

```
const person = ["John", "Doe", 46];
```

- person[0] returns John

The below is an object

```
const person = { firstName:"John",
                 lastName:"Doe",
                 age:46 };
```

```
person.lastName returns Doe
```

- Notice the { } not [ ]

Array Elements Can Be Objects

```
myArray[0] = Date.now;
myArray[1] = myFunction;
myArray[2] = myCars;
```

Array Properties and Methods

```
cars.length   // Returns the number of elements
cars.sort()   // Sorts the array
```

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let length = fruits.length;
```

```
// Access last element
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fruit = fruits[fruits.length - 1];
```

Add new element

- Use the push() method

```
const fruits =
["Banana", "Orange", "Apple"];


// Adds a new element (Lemon) to fruits
fruits.push("Lemon");
```

Add new element (2)

- Use length

```
const fruits = ["Banana", "Orange",
"Apple"];

fruits[fruits.length] = "Lemon";
// Adds "Lemon" to fruits
```

Be carefull

- Adding elements with high indexes can create undefined "holes" in an array

```
const fruits = ["Banana", "Orange",
"Apple"];

fruits[6] = "Lemon";  // Creates
undefined "holes" in fruits
```

Common error

```
const points = [40];

is different from

const points = new Array(40);
```

toString

- The JavaScript method toString() converts an array to a string of (comma separated) array values.

```
const fruits =
["Banana", "Orange", "Apple", "Mang
o"];

const str  = fruits.toString();
```

join

- The join() method also joins all array elements into a string.
- It behaves just like toString(), but in addition you can specify the separator

```
const fruits =
["Banana", "Orange", "Apple", "Mango"];

const msg = fruits.join(" * ");
```

pop

- The pop() method removes the last element from an array
  - It returns the element that was popped

```
const fruits =
["Banana", "Orange", "Apple", "Mango"];

fruits.pop();
```

push

- The push() method adds a new element to an array (at the end)
  - It returns the array's new length

```
const fruits =
["Banana", "Orange", "Apple", "Mango"];

fruits.push("Kiwi");
```

shift

- The shift() method removes the first array element and "shifts" all other elements to a lower index
  - It returns the value that was shifted

```
const fruits =
["Banana", "Orange", "Apple", "Mango"];

fruits.shift();
```

unshift

- The unshift() method adds a new element to an array (at the beginning), and "unshifts" older elements
  - The unshift() method returns the new array length.

```
const fruits =
["Banana", "Orange", "Apple", "Mango"];
fruits.unshift("Lemon");
```

delete

- Array elements can be deleted using the JavaScript operator `delete`.
- Using delete leaves undefined holes in the array.
- Use pop() or shift() instead.

```
const fruits =
["Banana", "Orange", "Apple", "Mango"];
delete fruits[0];
```

Merging arrays

- The concat() method **creates a new array** by merging (concatenating) existing arrays:
- The concat() method does not change the existing arrays. It always returns a new array.
- The concat() method can take any number of array arguments

```
const arr1 = ["Cecilie", "Lone"];
const arr2 = ["Emil", "Tobias", "Linus"];
const arr3 = ["Robin", "Morgan"];

const arr4 = arr1.concat(arr2);
const arr5 = arr1.concat(arr2, arr3);
```

Merging arrays (2)

- The concat() method can also take strings as arguments

```
const arr1 = ["Emil", "Tobias", "Linus"];
const myChildren = arr1.concat("Peter");
```

splice

- The splice() method adds new items to an array.

```
const fruits =
["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 0, "Lemon", "Kiwi");
```

- The first parameter (2) defines the position where new elements should be added (spliced in).
- The second parameter (0) defines how many elements should be removed.
- The rest of the parameters ("Lemon" , "Kiwi") define the new elements to be added.
- The splice() method **returns an array** with the deleted items

Using splice to Remove Elements

```
const fruits =
["Banana", "Orange", "Apple", "Mango"];

fruits.splice(0, 1);
```

- The first parameter (0) defines the position where new elements should be added (spliced in).
- The second parameter (1) defines how many elements should be removed.
- The rest of the parameters are omitted. No new elements will be added.

slice

- The slice() method slices out a piece of an array **into a new array**.

```
const fruits =
["Banana", "Orange", "Lemon", "Apple",
 "Mango"];
const citrus = fruits.slice(1);
```

- This example slices out a part of an array starting from array element 1 ("Orange")

slice (2)

- The slice() method **creates a new array**.
- The slice() method does not remove any elements from the source array.

```
const fruits =
["Banana", "Orange", "Lemon", "Apple",
 "Mango"];
const citrus = fruits.slice(3);
```

slice (3)

- The slice() method can take two arguments like slice(1, 3).
- The method then selects elements from the start argument, and up to (**but not including**) the end argument.
- If the end argument is omitted, like in the first examples, the slice() method slices out the rest of the array.

```
const fruits =
["Banana", "Orange", "Lemon", "Apple", "M
ango"];
const citrus = fruits.slice(1, 3);
```

sort

- The sort() method sorts an array **alphabetically**:

```
const fruits =
["Banana", "Orange", "Apple", "Mango"];
fruits.sort();
```

reverse

- The reverse() method reverses the elements **in an array**.

```
const fruits =
["Banana", "Orange", "Apple", "Mango"];
fruits.reverse();
```

- You can use it to sort an array in descending order:

```
const fruits =
["Banana", "Orange", "Apple", "Mango"];
fruits.sort();
fruits.reverse();
```

Numeric sort

- By default, the sort() function sorts values as strings.
- This works well for strings ("Apple" comes before "Banana").
- However, if numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1".
- Because of this, the sort() method will produce incorrect result when sorting numbers.

Numeric sort (2)

- sort can take a compare function
- For integers, a compare function should perform "integer comparison"

```
if num1 < num2 returns negative
  else if num1 > num2 returns positive
    else return 0
```

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a - b});
```

Numeric sort (3)

```
const points = [40, 100, 1, 5, 25, 10];

points.sort(fun);

function fun(a, b){return b - a}
```

Min and Max

- How can you find min and max using previous functions?

indexOf

- The indexOf() method searches an array for an element value and returns its position.

```
const fruits =
["Apple", "Orange", "Apple", "Mango"];
let position = fruits.indexOf("Apple");
```

- Returns -1 if not found

lastIndexOf

- Array.lastIndexOf() is the same as Array.indexOf(), but returns the position of the last occurrence of the specified element.

```
const fruits =
["Apple", "Orange", "Apple", "Mango"];
let position =
fruits.lastIndexOf("Apple") + 1;
```

forEach

- The forEach() method calls a function (a callback function) once for each array element.

```
const numbers = [45, 4, 9, 16, 25];
let txt = "";
numbers.forEach(myFunction);

function myFunction(value, index, array) {
  txt += value + " ";
}
```

- Note that the function takes 3 arguments:
  - The item value
  - The item index
  - The array itself

forEach (2)

- Since the previous example uses only the value parameter. It can be rewritten to:

- ```
  const numbers = [45, 4, 9, 16, 25];
  let txt = "";
  numbers.forEach(myFunction);

  function myFunction(value) {
    txt += value + " ";
  }
  ```


  ```

map

- The map() method **creates a new array** by performing a function on each array element.
- The map() method does not execute the function for array elements without values.
- The map() method does not change the original array.

map (2)

```
const numbers1 = [45, 4, 9, 16, 25];
const numbers2 =
numbers1.map(myFunction);

function myFunction(value, index, array)
{
  return value * 2;
}
```

filter

- The filter() method **creates a new array** with array elements that pass a test.

```
const numbers = [45, 4, 9, 16, 25];
const over18 =
numbers.filter(myFunction);

function myFunction(value, index,
array) {
  return value > 18;
}
```

reduce

- The reduce() method runs a function on each array element to produce (reduce it to) a single value.
- The reduce() method works from left-to-right in the array.
  - See also reduceRight().
- The reduce() method **does not reduce the original array**.

reduce (2)

```
const numbers = [45, 4, 9, 16, 25];
let sum =
numbers.reduce(myFunction);

function myFunction(total, value,
index, array) {
  return total + value;
}
```

reduce (3)

- The reduce() method can accept an initial value:

```
const numbers = [45, 4, 9, 16, 25];
let sum =
numbers.reduce(myFunction, 100);

function myFunction(total, value) {
  return total + value;
}
```

every

- The every() method checks if all array values pass a test.

```
const numbers = [45, 4, 9, 16, 25];
let allOver18
= numbers.every(myFunction);

function myFunction(value, index,
array) {
  return value > 18;
}
```

some

- The some() method checks if some array values pass a test.

```
const numbers = [45, 4, 9, 16, 25];
let someOver18 =
numbers.some(myFunction);

function myFunction(value, index,
array) {
  return value > 18;
}
```

find

- The find() method returns the value of the first array element that passes a test function.

```
const numbers = [4, 9, 16, 25, 29];
let first = numbers.find(myFunction);

function myFunction(value, index,
array) {
  return value > 18;
}
```

Three ways to define a function

- Named function:

  function sum(a, b) { return a + b;}

- Anonymous function:

  (function (a, b) { return a + b ;})

- Arrow functions
  - They have some limitations

Arrow function

- An arrow function expression is a compact alternative to a traditional function expression

```
let f = (param1, param2) => { body }
```

- The parentheses are optional with one single parameter

```
param => {
const a = 1;
return a + param;
}
```

// Traditional Anonymous Function

```
(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 if body has one statement.

```
(a) => a + 100;
```

// 3. Remove the argument parentheses

```
a => a + 100;
```

Destructuring arrays (3)

```
let a, b, rest;
[a, b] = [10, 20];

console.log(a);
// expected output: 10

console.log(b);
// expected output: 20
```

Destructuring arrays (2)

```
let a, b, rest;
[a, b] = [10, 20];

console.log(a);
// expected output: 10

console.log(b);
// expected output: 20

[a, b, ...rest] = [10, 20, 30, 40, 50];

console.log(rest);
// expected output: Array [30,40,50]
```

Destructuring arrays

```
let a, b, rest;
[a, b] = [10, 20];

console.log(a);
// expected output: 10

console.log(b);
// expected output: 20

[a, b, ...rest] = [10, 20, 30, 40, 50];

console.log(rest);
// expected output: Array [30,40,50]

const arr2 = [...rest, a, b, ...rest];

console.log(arr2);
```