Learn JavaScript: Arrays and Iteration

Property `.length`

The `.length` property of a JavaScript array indicates the number of elements the array contains.

```
const numbers = [1, 2, 3, 4];

numbers.length // 4
```

Index

Array elements are arranged by *index* values, starting at `0` as the first element index. Elements can be accessed by their index using the array name, and the index surrounded by square brackets.

```
// Accessing an array element
const myArray = [100, 200, 300];

console.log(myArray[0]); // 100
console.log(myArray[1]); // 200
console.log(myArray[2]); // 300
```

Method `.push()`

The `.push()` method of JavaScript arrays can be used to add one or more elements to the end of an array. `.push()` mutates the original array returns the new length of the array.

```
// Adding a single element:
const cart = ['apple', 'orange'];
cart.push('pear');

// Adding multiple elements:
const numbers = [1, 2];
numbers.push(3, 4, 5);
```

Method `.pop()`

The `.pop()` method removes the last element from an array and returns that element.

```
const ingredients = ['eggs', 'flour',
'chocolate'];

const poppedIngredient
= ingredients.pop(); // 'chocolate'
console.log(ingredients); // ['eggs',
'flour']
```

Mutable

JavaScript arrays are *mutable*, meaning that the values they contain can be changed.

Even if they are declared using `const` , the contents can be manipulated by reassigning internal values or using methods like `.push()` and `.pop()` .

```
const names = ['Alice', 'Bob'];

names.push('Carl');
// ['Alice', 'Bob', 'Carl']
```

Arrays

Arrays are lists of ordered, stored data. They can hold items that are of any data type. Arrays are created by using square brackets, with individual elements separated by commas.

```
// An array containing numbers
const numberArray = [0, 1, 2, 3];

// An array containing different data
types
const mixedArray = [1, 'chicken', false];
```

While Loop

The `while` loop creates a loop that is executed as long as a specified condition evaluates to `true` . The loop will continue to run until the condition evaluates to `false` . The condition is specified before the loop, and usually, some variable is incremented or altered in the `while` loop body to determine when the loop should stop.

```
while (condition) {
  // code block to be executed
}

let i = 0;

while (i < 5) {
  console.log(i);
  i++;
}
```

Reverse Loop

A `for` loop can iterate "in reverse" by initializing the loop variable to the starting value, testing for when the variable hits the ending value, and decrementing (subtracting from) the loop variable at each iteration.

```
const items = ['apricot', 'banana',
'cherry'];

for (let i = items.length - 1; i >= 0; i -
= 1) {
  console.log(`${i}. ${items[i]}`);
}

// Prints: 2. cherry
// Prints: 1. banana
// Prints: 0. apricot
```

Do...While Statement

A `do...while` statement creates a loop that executes a block of code once, checks if a condition is true, and then repeats the loop as long as the condition is true. They are used when you want the code to always execute at least once. The loop ends when the condition evaluates to false.

```
x = 0
i = 0

do {
  x = x + i;
  console.log(x)
  i++;
} while (i < 5);

// Prints: 0 1 3 6 10
```

For Loop

A `for` loop declares looping instructions, with three important pieces of information separated by semicolons `;` :

- The *initialization* defines where to begin the loop by declaring (or referencing) the iterator variable
- The *stopping condition* determines when to stop looping (when the expression evaluates to `false`)
- The *iteration statement* updates the iterator each time the loop is completed

```
for (let i = 0; i < 4; i += 1) {
  console.log(i);
};

// Output: 0, 1, 2, 3
```

Looping Through Arrays

An array's length can be evaluated with the `.length` property. This is extremely helpful for looping through arrays, as the `.length` of the array can be used as the stopping condition in the loop.

```
for (let i = 0; i < array.length; i++){
  console.log(array[i]);
}

// Output: Every item in the array
```

Break Keyword

Within a loop, the `break` keyword may be used to exit the loop immediately, continuing execution after the loop body.

Here, the `break` keyword is used to exit the loop when `i` is greater than 5.

```
for (let i = 0; i < 99; i += 1) {
  if (i > 5) {
     break;
  }
  console.log(i)
}

// Output: 0 1 2 3 4 5
```

Nested For Loop

A nested `for` loop is when a `for` loop runs inside another `for` loop.
The inner loop will run all its iterations for *each* iteration of the outer loop.

```
for (let outer = 0; outer < 2; outer += 1)
{
  for (let inner = 0; inner < 3; inner +=
1) {
    console.log(`${outer}-${inner}`);
  }
}

/*
Output:
0-0
0-1
0-2
1-0
1-1
1-2
*/
```

Loops

A *loop* is a programming tool that is used to repeat a set of instructions. *Iterate* is a generic term that means “to repeat” in the context of *loops*. A *loop* will continue to *iterate* until a specified condition, commonly known as a *stopping condition*, is met.

Functions Assigned to Variables

In JavaScript, functions are a data type just as strings, numbers, and arrays are data types. Therefore, functions can be assigned as values to variables, but are different from all other data types because they can be invoked.

```
let plusFive = (number) => {
  return number + 5;
};
// f is assigned the value of plusFive
let f = plusFive;

plusFive(3); // 8
// Since f has a function value, it can be invoked.
f(9); // 14
```

Callback Functions

In JavaScript, a callback function is a function that is passed into another function as an argument. This function can then be invoked during the execution of that higher order function (that it is an argument of).
Since, in JavaScript, functions are objects, functions can be passed as arguments.

```
const isEven = (n) => {
  return n % 2 == 0;
}

let printMsg = (evenFunc, num) => {
  const isNumEven = evenFunc(num);
  console.log(`The number ${num} is an even number: ${isNumEven}.`)
}

// Pass in isEven as the callback function
printMsg(isEven, 4);
// Prints: The number 4 is an even number: True.
```

Higher-Order Functions

In Javascript, functions can be assigned to variables in the same way that strings or arrays can. They can be passed into other functions as parameters or returned from them as well.
A "higher-order function" is a function that accepts functions as parameters and/or returns a function.

Array Method `.reduce()`

The `.reduce()` method iterates through an array and returns a single value.
It takes a callback function with two parameters `(accumulator, currentValue)` as arguments. On each iteration, `accumulator` is the value returned by the last iteration, and the `currentValue` is the current element. Optionally, a second argument can be passed which acts as the initial value of the accumulator.
Here, the `.reduce()` method will sum all the elements of the array.

```
const arrayOfNumbers = [1, 2, 3, 4];

const sum
= arrayOfNumbers.reduce((accumulator,
currentValue) => {
  return accumulator + currentValue;
});

console.log(sum); // 10
```

Array Method `.forEach()`

The `.forEach()` method executes a callback function on each of the elements in an array in order.
Here, the callback function containing a `console.log()` method will be executed `5` times, once for each element.

```
const numbers = [28, 77, 45, 99, 27];

numbers.forEach(number => {
  console.log(number);
});
```

Array Method `.filter()`

The `.filter()` method executes a callback function on each element in an array. The callback function for each of the elements must return either `true` or `false`. The returned array is a new array with any elements for which the callback function returns `true`.
Here, the array `filteredArray` will contain all the elements of `randomNumbers` but `4`.

```
const randomNumbers = [4, 11, 42, 14, 39];
const filteredArray
= randomNumbers.filter(n => {
  return n > 5;
});
```

Array Method `.map()`

The `.map()` method executes a callback function on each element in an array. It returns a new array made up of the return values from the callback function.
The original array does not get altered, and the returned array may contain different elements than the original array.

```
const finalParticipants = ['Taylor',
'Donald', 'Don', 'Natasha', 'Bobby'];

const announcements
= finalParticipants.map(member => {
  return member + ' joined the contest.';
})

console.log(announcements);
```