**Array built-in methods**

**array.pop()**

The pop() method removes the last element from an array and returns that element. This method changes the length of the array.

const plants = ['broccoli', 'cauliflower', 'cabbage', 'kale', 'tomato'];

console.log(plants.pop());

// Expected output: "tomato"

console.log(plants);

// Expected output: Array ["broccoli", "cauliflower", "cabbage", "kale"]

plants.pop();

console.log(plants);

// Expected output: Array ["broccoli", "cauliflower", "cabbage"]

**array.push()**

The push() method adds one or more elements to the end of an array and returns the new length of the array.

const animals = ['pigs', 'goats', 'sheep'];

const count = animals.push('cows');

console.log(count);

// Expected output: 4

console.log(animals);

// Expected output: Array ["pigs", "goats", "sheep", "cows"]

animals.push('chickens', 'cats', 'dogs');

console.log(animals);

// Expected output: Array ["pigs", "goats", "sheep", "cows", "chickens", "cats", "dogs"]

**array.slice()**

The slice() method returns a shallow copy of a portion of an array into a new array object selected from start to end (end not included) where start and end represent the index of items in that array. The original array does not get modified.

const animals = ['ant', 'bison', 'camel', 'duck', 'elephant'];

console.log(animals.slice(2));

// Expected output: Array ["camel", "duck", "elephant"]

console.log(animals.slice(2, 4));

// Expected output: Array ["camel", "duck"]

console.log(animals.slice(1, 5));

// Expected output: Array ["bison", "camel", "duck", "elephant"]

**array.splice();**

The splice() method changes the contents of an array by removing or replacing existing elements and/or adding new elements in their place.

const months = ['Jan', 'March', 'April', 'June'];

months.splice(1, 0, 'Feb');

// inserts at index 1

console.log(months);

// Expected output: Array ["Jan", "Feb", "March", "April", "June"]

months.splice(4, 1, 'May');

// replaces 1 element at index 4

console.log(months);

// Expected output: Array ["Jan", "Feb", "March", "April", "May"]

**string.indexOf()**

The indexOf() method returns the index within the string of the first occurrence of the specified value, starting the search at fromIndex. It returns -1 if the value is not found.

const paragraph = 'The quick brown fox jumps over the lazy dog. If the dog barked, was it really lazy?';

const searchTerm = 'dog';

const indexOfFirst = paragraph.indexOf(searchTerm);

console.log(`The index of the first "${searchTerm}" from the beginning is ${indexOfFirst}`);

// expected output: "The index of the first "dog" from the beginning is 40"

console.log(`The index of the 2nd "${searchTerm}" is ${paragraph.indexOf(searchTerm, (indexOfFirst + 1))}`);

// expected output: "The index of the 2nd "dog" is 52"

**string.split()**

The split() method divides a string into an ordered list of substrings, puts these substrings into an array, and returns the array. The division is done by searching for a pattern; where the pattern is provided as the first parameter in the method's call.

const str = 'The quick brown fox jumps over the lazy dog.';

const words = str.split(' ');

console.log(words[3]);

// expected output: "fox"

const chars = str.split('');

console.log(chars[8]);

// expected output: "k"

const strCopy = str.split();

console.log(strCopy);

// expected output: Array ["The quick brown fox jumps over the lazy dog."]

**string.slice()**

The slice() method extracts a section of a string and returns it as a new string without modifying the original string.

const str = 'The quick brown fox jumps over the lazy dog.';

console.log(str.slice(31));

// Expected output: "the lazy dog."

console.log(str.slice(4, 19));

// Expected output: "quick brown fox"

console.log(str.slice(-4));

// Expected output: "dog."

console.log(str.slice(-9, -5));

// Expected output: "lazy"

**String template literals**

Template literals are strings that allow for directly embedded expressions. They begin and end with backticks (`your string here`) and you can embed any data type within the string using ${put your data here}. This is very useful for "interpolating" (embedding) large quantities of data inside strings.

const firstName = 'Rachel';

const greeting = `Hello, ${firstName}!`;

console.log(greeting);

//Expected output: Hello, Rachel!

const studentObj = {

name: 'Rachel',

class: 'Intro to Programming'

};

const studentInfo = `The student's name is ${studentObj.name}, and they are taking ${studentObj.class}.`;

console.log(studentInfo);

//Expected output: The student's name is Rachel, and they are taking Intro to Programming.

**Function styles**

**Anonymous**

An anonymous function is a function without a function name. Only function expressions can be anonymous. Function declarations must have a name. Anonymous functions are useful when you use the function only once, as they are faster to write.

// When used as a function expression

function () {//do stuff};

// Or using ES6 arrow notation

() => {//do stuff};

**Named**

A named function is a function with a function name. We name our functions when we need to re-use them later on in our code.

// Function declaration

function foo() {};

// Named function expression

(function bar() {});

// or using the ECMAScript 2015 arrow notation

const foo = () => {};

**Fat Arrow**

An arrow function expression is a compact alternative to the traditional function expression but is limited and can't be used in all situations. They are inherently anonymous but can be stored in variables and re-used.

//With one parameter

param1 => {do stuff}

//With multiple parameters

(param1, param2) => {do stuff};

//With multiple parameters stored in a variable

var myFunction = (param1, param2) => {do stuff};

//Now a more complex example with an object

const materials = [

'Hydrogen',

'Helium',

'Lithium',

'Beryllium'

];

console.log(materials.map(material => material.length));

// Expected output: Array [8, 6, 7, 9]

**ES6 module pattern**

**Export**

The export statement is used when creating JavaScript modules to export functions, objects, or primitive values from the module file into other files or programs. Other programs then use the import statement to import the module.

//You can export functions, classes, and variables individually.

export var name1, name2, …, nameN;

export function functionName(){...}

export class ClassName {...}

//Or you can export functions, etc in a list.

function sayHi(name) {

alert(`Hello, ${name}!`);

}

function sayBye(name) {

alert(`Bye, ${name}!`);

}

export {sayHi, sayBye}; // a list of exported variables

**Import**

The import statement is used to import read-only functions exported by another module.

//You can import functions individually.

import {sayHi, sayBye} from './say.js';

sayHi('Dave'); // Hello, Dave!

sayBye('John'); // Bye, Dave!

//or, if you use a lot of functions in the module file, you can import all of them using \*.

import \* as say from './say.js';

say.sayHi('Dave');

say.sayBye('Dave');

//If you only use a few functions from a module file, don't use the \* command

You'll get the answer when you get to webpacking.

[SOURCES: MDN](https://developer.mozilla.org/en-US/)

Sources:

MITxPro BootCamp

https://executive-ed.xpro.mit.edu/professional-certificate-coding?utm\_source=Google&utm\_medium=c&utm\_term=%2Bmit%20%2Bcoding&utm\_location=9007574&utm\_campaign=B-365D\_US\_GG\_SE\_PCC\_Brand&utm\_content=MIT-Coding\_\_\_School\_Duration&gclid=Cj0KCQjw3duCBhCAARIsAJeFyPVYZJwvrGBn\_93CMgrokibm87uQ\_OnDXxiBDYwzsRiQcPqa2\_vVawoaAiLCEALw\_wcB