Lesson 04.01

arrays - index, array.length

const vs. let

array methods: push(), sort(), pop(),

nested / 2D arrays

arrays

Arrays are variables that can hold more than one value at a time. Arrays have a datatype of **object**. Here are some key things to know about arrays:

- array items exist as a list, surrounded by square brackets: **[item1, item2, item3]**
- each item is assigned a position number, or **index**, with the first item at index 0
- array items can be of any data type
- items of different data types can be in the same array: **['apples', 4, true, 3.5]**

using const to declare an array

As we have learned, primitive variables (strings, numbers, booleans) declared with **const** cannot be modified (changed) in any way.

1. Declare a constant with **const**, and try to change it. We get an error:

```
const DOZEN = 12;
DOZEN = 13; // ERROR: Assigment to constant variable
```

const for objects

Unlike primitives, arrays declared with **const** *can* be modified--to an extent. Items can be changed, added or deleted, but the array bject itself cannot be **mutated** to a different datatype. So, with **const**, once an array, always an array.

To declare an array, start with **let** or **const**, followed by a name. We will start with **let** in order to later demonstrate our point about mutability. The array value consists of a pair square brackets containing items separated by commas.

It is a good practice to pluralize array names: **fruits** not **fruit**. Adding **Arr** further clarifies that this is an array: **fruitsArr**.

2. Declare an array called **fruitsArr**, with three items:

```
let fruitsArr = ['apple', 'banana', 'cherry'];
console.log(fruitsArr);
```

3. Check the Console. Expand the arrow to see the numbered items, with 'apple' at index 0.

index

Array items are stored in a numeric order, called **index**.

- The first item in an array is at index 0 the second item is at index 1, and so on.
- Use square bracket syntax to get individual array items: **array[0]**

4. Get the first item at index 0 and the last item at index 2:

```
console.log(fruitsArr[0]); // apple
console.log(fruitsArr[2]); // cherry
```

setting array values by index

To set the value of an array item, refer to it by index:

5. Replace the first item, 'apple', with 'peach':

```
fruitsArr[0] = 'peach';
console.log(fruitsArr);
// ['peach', 'banana', 'cherry']
```

One way to add an item at the end of the array is to assign it by index:

6. Add a fruit to the end of the array, at index 3:

```
fruitsArr[3] = 'mango';
console.log(fruitsArr);
// ['peach', 'banana', 'cherry', 'mango']
```

mutating an array (changing its datatype)

A downside of **let** for arrays is that it does not protect the data type. You could inadvertently change the array to a string or any other datatype. Use 'const' to prevent such a mistake from occurring.

7. Change fruitsArr to a string. Just like that, no more array:

```
fruitsArr = "oops";
console.log(fruitsArr); // oops
```

8. Declare another fruits array, this time with 'const'. We need a new name, as we cannot redeclare an existing 'const' or 'let':

```
const fruits = ['apricot', 'pear', 'kiwi', 'grape'];
```

9. Try to mutate the array into a string. You get an error:

```
fruits = "whoops";
// ERROR: Assignment to constant variable
```

array.length

The **length** property of an array returns the number of items in the array.

10. Get the length of the array:

```
console.log(fruits.length); // 4
```

array.length - 1

- The first item in an array is at index 0.
- **array[0]** gets the first item.
- The last item is at index **array.length - 1**.
- **array[arrray.length - 1]** gets the last item.

11. Save the last item in fruits to a variable:

```
let lastFruit = fruits[fruits.length - 1];
console.log(lastFruit); // grape
```

12. Try that again, but without subtracting 1. We get 'undefined', because there is no index 4. Indexing starts at 0, so the last of the 4 items is at index 3:

```
lastFruit = fruits[fruits.length];
console.log(lastFruit); // undefined
```

random array items

To select an item at random from an array:

- generate a random integer from 0 to array.length-1
- pass that number to the array square brackets

13. Get a random fruit from fruits:

```
let r = Math.floor(Math.random() * fruits.length);
console.log(r, fruits[r]); // 2 kiwi
```

Run it a few times to see that the fruit keeps changing.

array methods

The array object has numerous methods, some of which we will learn about now:

array.push(item)

The 4th and last item of the fruits array is at index 3. Therefore, a 5th item would occupy index 4.

14. Add a 5th fruit to the array:

```
fruits[4] = 'banana';
console.log(fruits); // ['apricot', 'pear', 'kiwi', 'grape', 'banana']
```

That worked, but hard-coding numbers is to be avoided, since the length of the array has just changed. Next time, the new item would be at index 5, and so on. More dynamically, we could use **array.length** as the index.

15. Add a 6th and 7th fruit using **array.length** as the dynamic index of the new, last item:

```
fruits[fruits.length] = 'pineapple';
console.log(fruits); // ['apricot', 'pear', 'kiwi', 'grape', 'banana',
'pineapple']
fruits[fruits.length] = 'papaya';
console.log(fruits); // ['apricot', 'pear', 'kiwi', 'grape', 'banana',
'pineapple', 'papaya']
```

With fruits.length, we avoid hard-coding the index as we add more items. But there's yet a better way: array.push(item)

The **push()** method takes one or more items as its argument and adds it/them to the end of the array.

16. Using the **push()** method, add 'lime' to the end of the array.

```
fruits.push('lime');
console.log(fruits); // ['apricot', 'pear', 'kiwi', 'grape', 'banana',
'pineapple', 'papaya', 'lime]
```

17. Push more than one item at a time:

```
fruits.push('lemon', 'blueberry');
console.log(fruits); // ['apricot', 'pear', 'kiwi', 'grape', 'banana',
'pineapple', 'papaya', 'lime', 'lemon', 'blueberry']
```

array.sort()

Called on an array of strings, the **sort()** method puts the items in alphabetical order. Sort does not return (make) a new array; it operates in place on the existing array:

18. Sort the fruits array:

```
fruits.sort();
console.log(fruits);
// ['apricot',  'banana', 'blueberry', 'grape' 'kiwi',  'lemon',
'lime', 'papaya', 'pear', 'pineapple']
```

array.pop()

The **pop()** method removes the last item from the array. It also returns the item, so it can be saved to a variable.

19. Pop the last item from the array. Log the array to see that 'pineapple' is gone:

```
fruits.pop();
console.log(fruits);
// ['apricot',  'banana', 'blueberry', 'grape' 'kiwi',  'lemon',
'lime', 'papaya', 'pear']
```

20. Pop again, but this time save the popped item to a variable and then log it:

```
let poppedItem = fruits.pop();
console.log(poppedItem); // pear
console.log(fruits);
// ['apricot',  'banana', 'blueberry', 'grape' 'kiwi',  'lemon',
'lime', 'papaya']
```

declaring an empty array

An empty array is declared as just a pair of empty square brackets--no items.

21. Declare a new, empty array:

```
const veggies = [];
```

22. Push a few items into the array:

```
veggies.push('carrot', 'beet', 'arugula', 'cucumber');
veggies.push('kale', 'broccoli', 'lettuce', 'celery');
console.log(veggies); // ['carrot', 'beet', 'arugula', 'cucumber',
'kale', 'broccoli', 'lettuce', 'celery']
```

23. Sort the veggies array:

```
fruits.sort();
console.log(veggies); // ['arugula', 'beet', 'broccoli', 'carrot',
'celery', 'cucumber', 'kale', 'lettuce']
```

arrays of numbers

The simple sort() method, without arguments, works on strings and so regards numbers as strings and alphabetizes them. As a string, "1" is less than "2", as expected, but "100" is also less than "2", just as "app" is less than "b".

24. Declare an array of numbers:

```
const nums = [3, 5, 220, 2, 17, 105, 45, 65, 1008, 25];
```

25. Sort the nums array. Rather than being put in numeric order, the items are alphabetized:

```
nums.sort();
console.log(nums); // [1008, 105, 17, 2, 220, 25, 3, 45, 5, 65]
```

sort callback function

Sorting an array of numbers requires that a callback be passed to the sort method. Recall from a previous lesson that a callback is a function passed to another function as its argument.

- The callback takes two arguments of its own: a and b.
- The callback returns a - b.
- Under the hood, the method compares a to b, and if necessary, swaps their position.
- Working iteratively (again and again on a loop), this compare-and-swap procedure continues until there are no more swaps to be made, meaning that the items are sorted.

26. Call the sort method on the nums array, passing in the callback function. The result is properly numbers:

```
nums.sort(function(a,b) {
    return b - a; // b - a is descending
});
console.log(nums); // [2, 3, 5, 17, 25, 45, 65, 105, 220, 1008]
```

array.reverse()

The reverse method reverses the order of array items, so if they are first sorted, we get a reverse sort.

27. Reverse the fruits array, which is already sorted in alphabetical order:

```
fruits.reverse();
console.log(fruits); // ['papaya', 'lime', 'lemon', 'kiwi', 'grape',
'blueberry', 'banana','apricot']
```

The reverse() method can also be called on the numbers array once it is sorted, but it can also be sorted in reverse to begin with by returning **b - a** instead of **a - b**:

28. Push a few numbers into **nums**, so that it is no longer in ascending order, and then sort the array in descending (reverse) order:

```
nums.push(21, 325, 1234, 7, 88, 4);
console.log(nums); // [2, 3, 5, 17, 25, 45, 65, 105, 220, 1008, 21,
325, 1234, 7, 88, 4]
```

29. Sort the array in descending (reverse) order by flipping the return statement to **b - a**:

```
nums.sort(function(a,b) {
    return b - a; // b - a is descending
});
console.log(nums); // [1234, 1008, 325, 220, 105, 88, 65, 45, 25, 21,
17, 7, 5, 4, 3, 2]
```

2D (nested) arrays

An array can have arrays for its items. The terms to describe such an array include: **matrix, 2D array** and **nested array**

30. Make a 2x4 array of two items, each an array of four items:

```
const twoByFour = [[1,2,3,4], [5,6,7,8]];
console.log(twoByFour); // [1,2,3,4], [5,6,7,8]
```

```
console.log(twoByFour.length); // 2
console.log(twoByFour[0]); // [1,2,3,4]
console.log(twoByFour[1]); // [5,6,7,8]
```

To access individual number from this array of arrays, use double-square brackets.

31. Get the first and last items from each inner array:

```
console.log(nestedArr[0][0]); // 1
console.log(nestedArr[0][3]); // 4
console.log(nestedArr[1][0]); // 5
console.log(nestedArr[1][3]); // 8
```

32. Make a 3x3 tic-tac-toe "board", where all 9 squares start with a value of **null**:

```
let ticTacToe = [
    [null, null, null],
    [null, null, null],
    [null, null, null]
];
```

33. Game on! **X** chooses the middle, and **O** chooses the lower left square:

```
ticTacToe[1][1] = "X";
ticTacToe[2][0] = "O";
```

34. Log **ticTacToe** to see the game progress:

```
console.log(ticTacToe);
/*
(3) [Array(3), Array(3), Array(3)]
0: (3) [null, null, null]
1: (3) [null, 'X', null]
2: (3) ['O', null, null]
*/
```