Javascript Day - 5

Functions(contd.), Array methods

Function Expressions

There is another syntax for creating a function that is called a Function Expression.

It allows us to create a new function in the middle of any expression.

```
1  let sayHi = function() {
2    alert( "Hello" );
3  };
```

Here we can see a variable sayHi getting a value, the new function, created as function() { alert("Hello"); }

Please note, there's no name after the function keyword. Omitting a name is allowed for Function Expressions.

Function Expression

We can copy a function to another variable:

```
1  function sayHi() {   // (1) create
2    alert( "Hello" );
3  }
4
5  let func = sayHi;    // (2) copy
6
7  func(); // Hello     // (3) run the copy (it works)!
8  sayHi(); // Hello    //     this still works too (why wouldn't it)
```

Callback Function

Function which is passed to another function as an argument is called callback function.

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

function showOk() {
  alert( "You agreed." );
}

function showCancel() {
  alert( "You canceled the execution." );
}

// usage: functions showOk, showCancel are passed as arguments to ask
ask("Do you agree?", showOk, showCancel);
```

The arguments showOk and showCancel of ask are called callback functions or just callbacks.

Callback Function

We can use Function Expressions to write an equivalent, shorter function:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask(
  "Do you agree?",
  function() { alert("You agreed."); },
  function() { alert("You canceled the execution."); }
);
```

Here, functions are declared right inside the ask(...) call. They have no name, and so are called anonymous. Such functions are not accessible outside of ask (because they are not assigned to variables), but that's just what we want here.

Function Expression vs Function Declaration

1. Function declarations are standalone,

   Function expressions are created inside an expression or inside another syntax construct.

2. A Function Expression is created when the execution reaches it and is usable only from that moment

   A Function Declaration can be called earlier than it is defined.

Summary

- Functions are values. They can be assigned, copied or declared in any place of the code.
- If the function is declared as a separate statement in the main code flow, that's called a "Function Declaration".
- If the function is created as a part of an expression, it's called a "Function Expression".
- Function Declarations are processed before the code block is executed. They are visible everywhere in the block.
- Function Expressions are created when the execution flow reaches them.

Arrow Function

There's another very simple and concise syntax for creating functions, that's often better than Function Expressions.

It's called "arrow functions", because it looks like this:

```
let func = (arg1, arg2, ..., argN) => expression;
```

This creates a function func that accepts arguments arg1..argN, then evaluates the expression on the right side with their use and **returns its result**.

```
let func = function(arg1, arg2, ..., argN) {
  return expression;
};
```

Multiline Arrow Function

Sometimes we need a more complex function, with multiple expressions and statements. In that case, we can enclose them in curly braces. The major difference is that curly braces require a return within them to return a value (just like a regular function does).

```
let sum = (a, b) => {  // the curly brace opens a multiline function
  let result = a + b;
  return result; // if we use curly braces, then we need an explicit "return"
};

alert( sum(1, 2) ); // 3
```

Array Methods

Adding or removing elements

1. splice
2. slice
3. concat

Iterate: forEach

Searching in Array

Transforming an Array

Array.isArray()

Array Methods - Adding or Removing elements (splice)

The arr.splice method is a swiss army knife for arrays. It can do everything: insert, remove and replace elements.

```
1  arr.splice(start[, deleteCount, elem1, ..., elemN])
```

It modifies arr starting from the index start: removes deleteCount elements and then inserts elem1, ..., elemN at their place. Returns the array of removed elements.

Array Methods - Adding or Removing elements (splice)

```
let arr = ["I", "study", "JavaScript"];

arr.splice(1, 1); // from index 1 remove 1 element

alert( arr ); // ["I", "JavaScript"]
```

```
let arr = ["I", "study", "JavaScript", "right", "now"];

// remove 2 first elements
let removed = arr.splice(0, 2);

alert( removed ); // "I", "study" <-- array of removed elements
```

```
let arr = ["I", "study", "JavaScript", "right", "now"];

// remove 3 first elements and replace them with another
arr.splice(0, 3, "Let's", "dance");

alert( arr ) // now ["Let's", "dance", "right", "now"]
```

```
let arr = ["I", "study", "JavaScript"];

// from index 2
// delete 0
// then insert "complex" and "language"
arr.splice(2, 0, "complex", "language");

alert( arr ); // "I", "study", "complex", "language", "JavaScript"
```

Array Methods - Adding or Removing elements (splice)

Negative indexes allowed

```
let arr = [1, 2, 5];

// from index -1 (one step from the end)
// delete 0 elements,
// then insert 3 and 4
arr.splice(-1, 0, 3, 4);

alert( arr ); // 1,2,3,4,5
```

Array Methods - Adding or Removing elements (slice)

It returns a new array copying to it all items from index start to end (not including end). Both start and end can be negative, in that case position from array end is assumed.

```
arr.slice([start], [end])
```

It's similar to a string method str.slice, but instead of substrings it makes subarrays.

```
let arr = ["t", "e", "s", "t"];

alert( arr.slice(1, 3) ); // e,s (copy from 1 to 3)

alert( arr.slice(-2) ); // s,t (copy from -2 till the end)
```

We can also call it without arguments: arr.slice() creates a copy of arr. That's often used to obtain a copy for further transformations that should not affect the original array.

Array Methods - Adding or Removing elements (concat)

The method arr.concat creates a new array that includes values from other arrays and additional items.

```
arr.concat(arg1, arg2...)
```

It accepts any number of arguments – either arrays or values.

The result is a new array containing items from arr, then arg1, arg2 etc.

If an argument argN is an array, then all its elements are copied. Otherwise, the argument itself is copied.

```
let arr = [1, 2];

// create an array from: arr and [3,4]
alert( arr.concat([3, 4]) ); // 1,2,3,4

// create an array from: arr and [3,4] and [5,6]
alert( arr.concat([3, 4], [5, 6]) ); // 1,2,3,4,5,6

// create an array from: arr and [3,4], then add values 5 and 6
alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6
```

Array Methods - Iterate:forEach

The arr.forEach method allows to run a function for every element of the array.

```
arr.forEach(function(item, index, array) {
  // ... do something with item
});
```

```
// for each element call alert
["Bilbo", "Gandalf", "Nazgul"].forEach(alert);
```

```
["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {
  alert(`${item} is at index ${index} in ${array}`);
});
```

Array Methods - Searching in array

1. indexOf
2. lastIndexOf
3. Includes

The methods arr.indexOf and arr.includes have the similar syntax and do essentially the same as their string counterparts, but operate on items instead of characters:

- `arr.indexOf(item, from)` – looks for `item` starting from index `from`, and returns the index where it was found, otherwise `-1`.
- `arr.includes(item, from)` – looks for `item` starting from index `from`, returns `true` if found.

Usually these methods are used with only one argument: the `item` to search. By default, the search is from the beginning.

Array Methods - Searching in array

The previous methods works perfectly for normal value. But in case we want to find object with particular value we can use the following:

1. find
2. findIndex
3. findLastIndex
4. filter

Array Methods - Searching in array(find, findIndex, findLastIndex)

The function is called for elements of the array, one after another:

- item is the element.
- index is its index.
- array is the array itself.

If it returns true, the search is stopped, the item is returned. If nothing found, undefined is returned.

```
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];

let user = users.find(item => item.id == 1);

alert(user.name); // John
```

The arr.findIndex method has the same syntax, but returns the index where the element was found instead of th element itself. The value of `-1` is returned if nothing is found.

The arr.findLastIndex method is like `findIndex`, but searches from right to left, similar to `lastIndexOf`.

Array Methods - Searching in array(filter)

The find method looks for a single (first) element that makes the function return true.

If there may be many, we can use arr.filter(fn).

The syntax is similar to find, but filter returns an array of all matching elements:

```
let results = arr.filter(function(item, index, array) {
  // if true item is pushed to results and the iteration continues
  // returns empty array if nothing found
});
```

Array Methods - Searching in array(filter)

```
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];

// returns array of the first two users
let someUsers = users.filter(item => item.id < 3);

alert(someUsers.length); // 2
```

Array Methods - Transforming an Array

1. map
2. sort
3. reverse
4. split
5. join
6. reduce

Array Methods - Transforming an Array (map)

The arr.map method is one of the most useful and often used.

It calls the function for each element of the array and returns the array of results.

```
let result = arr.map(function(item, index, array) {
  // returns the new value instead of item
});
```

Array Methods - Transforming an Array (map)

For instance, here we transform each element into its length:

```
1  let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
2  alert(lengths); // 5,7,6
```

Array Methods - Transforming an Array (sort)

The call to arr.sort() sorts the array in place, changing its element order.

It also returns the sorted array, but the returned value is usually ignored, as arr itself is modified.

```
1  let arr = [ 1, 2, 15 ];
2
3  // the method reorders the content of arr
4  arr.sort();
5
6  alert( arr );  // 1, 15, 2
```

Array Methods - Transforming an Array (sort)

Did you notice anything strange in the outcome?

The order became 1, 15, 2. Incorrect. But why?

The items are sorted as strings by default.

Literally, all **elements are converted to strings for comparisons**. For strings, **lexicographic ordering** is applied and indeed "2" > "15".

Array Methods - Transforming an Array (sort)

To use our own sorting order, we need to supply a function as the argument of arr.sort().

The function should compare two arbitrary values and return:

```
function compare(a, b) {
  if (a > b) return 1; // if the first value is greater than the second
  if (a == b) return 0; // if values are equal
  if (a < b) return -1; // if the first value is less than the second
}
```

Array Methods - Transforming an Array (sort)

```
1  function compareNumeric(a, b) {
2    if (a > b) return 1;
3    if (a == b) return 0;
4    if (a < b) return -1;
5  }
6
7  let arr = [ 1, 2, 15 ];
8
9  arr.sort(compareNumeric);
10
11 alert(arr);  // 1, 2, 15
```

Array Methods - Transforming an Array (sort)

Actually, a comparison function is only required to return a positive number to say “greater” and a negative number to say “less”.

That allows to write shorter functions:

```
1  let arr = [ 1, 2, 15 ];
2
3  arr.sort(function(a, b) { return a - b; });
4
5  alert(arr);  // 1, 2, 15
```

Array Methods - Transforming an Array (reverse)

The method arr.reverse reverses the order of elements in arr.

```
1  let arr = [1, 2, 3, 4, 5];
2  arr.reverse();
3
4  alert( arr ); // 5,4,3,2,1
```

It also returns the array arr after the reversal.

Array Methods - Transforming an Array (split)

The str.split(delim) splits the string into an array by the given delimiter delim.

```
let names = 'Bilbo, Gandalf, Nazgul';

let arr = names.split(', ');
```

The split method has an optional second numeric argument – a limit on the array length. If it is provided, then the extra elements are ignored. In practice it is rarely used though:

```
let arr = 'Bilbo, Gandalf, Nazgul, Saruman'.split(', ', 2);
```

Array Methods - Transforming an Array (join)

The call arr.join(glue) does the reverse to split. It creates a string of arr items joined by glue between them.

```
let arr = ['Bilbo', 'Gandalf', 'Nazgul'];

let str = arr.join(';'); // glue the array into a string using ;

alert( str ); // Bilbo;Gandalf;Nazgul
```

Array Methods - Transforming an Array (reduce/reduceRight)

When we need to iterate over an array – we can use forEach, for or for..of.

When we need to iterate and return the data for each element – we can use map.

The methods arr.reduce and arr.reduceRight also belong to that breed, but are a little bit more intricate. They are used to calculate a single value based on the array.

```
let value = arr.reduce(function(accumulator, item, index, array) {
  // ...
}, [initial]);
```

Array Methods - Transforming an Array (reduce/reduceRight)

The function is applied to all array elements one after another and “carries on” its result to the next call.

Arguments:

- **accumulator** – is the result of the previous function call, equals initial the first time (if initial is provided).
- **item** – is the current array item.
- **index** – is its position.
- **array** – is the array.

Array Methods - Transforming an Array (reduce/reduceRight)

```
1  let arr = [1, 2, 3, 4, 5];
2
3  let result = arr.reduce((sum, current) => sum + current, 0);
4
5  alert(result); // 15
```

Array Methods - Transforming an Array (reduce/reduceRight)

We also can omit the initial value:

```
let arr = [1, 2, 3, 4, 5];

// removed initial value from reduce (no 0)
let result = arr.reduce((sum, current) => sum + current);

alert( result ); // 15
```

The result is the same. That's because if there's no initial, then reduce takes the first element of the array as the initial value and starts the iteration from the 2nd element.

The calculation table is the same as above, minus the first row.

But such use requires an extreme care. If the array is empty, then reduce call without initial value gives an error.

Array Methods - isArray

Arrays do not form a separate language type. They are based on objects.

So typeof does not help to distinguish a plain object from an array:

```
1  alert(typeof {}); // object
2  alert(typeof []); // object (same)
```

But arrays are used so often that there's a special method for that: Array.isArray(value). It returns true if the value is an array, and false otherwise.

```
1  alert(Array.isArray({})); // false
2
3  alert(Array.isArray([])); // true
```