JavaScript Day - 4

Switch, Arrays and Functions

Switch statement

A switch statement can replace multiple if checks.

It gives a more descriptive way to compare a value with multiple variants.

The switch has one or more case blocks and an optional default.

It looks like this:

```
switch(x) {
  case 'value1':  // if (x === 'value1')
    ...
    [break]

  case 'value2':  // if (x === 'value2')
    ...
    [break]

  default:
    ...
    [break]
}
```

Switch statement

The value of x is checked for a **strict equality** to the value from the first case (that is, value1) then to the second (value2) and so on.

If the equality is found, switch starts to execute the code starting from the corresponding case, until the nearest break (or until the end of switch).

If no case is matched then the default code is executed (if it exists).

```
switch(x) {
  case 'value1':  // if (x === 'value1')
    ...
    [break]

  case 'value2':  // if (x === 'value2')
    ...
    [break]

  default:
    ...
    [break]
}
```

Switch statement

Several variants of case which share the same code can be grouped.

```
let a = 3;

switch (a) {
  case 4:
    alert('Right!');
    break;

  case 3: // (*) grouped two cases
  case 5:
    alert('Wrong!');
    alert("Why don't you take a math class?");
    break;

  default:
    alert('The result is strange. Really.');
}
```

Array

Arrays are used to store ordered collections

1. Declaration
2. Get last elements with “at”
3. Methods pop/push, shift/unshift
4. Loops
5. Multidimensional arrays

Array - Declaration

There are two syntaxes for creating an empty array

```
1  let arr = new Array();
2  let arr = [];
```

Almost all the time, the second syntax is used. We can supply initial elements in the brackets:

```
1  let fruits = ["Apple", "Orange", "Plum"];
```

Array - Declaration

Array elements are numbered, starting with zero.

We can get an element by its number in square brackets:

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Plum
```

We can replace an element:

```
fruits[2] = 'Pear'; // now ["Apple", "Orange", "Pear"]
```

Or add a new element

```
fruits[3] = 'Lemon'; // now ["Apple", "Orange", "Pear", "Lemon"]
```

Array - Last Element of the array

It is somewhat similar to how we found the last character in a string.

We can explicitly calculate the last element index and then access it:

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits[fruits.length-1] ); // Plum
```

Luckily, there's a shorter syntax: fruits.at(-1)

```
let fruits = ["Apple", "Orange", "Plum"];

// same as fruits[fruits.length-1]
alert( fruits.at(-1) ); // Plum
```

Array - push/pop and shift/unshift

- **push** appends an element to the end.
- **shift** get an element from the beginning, advancing the queue, so that the 2nd element becomes the 1st.

- **unshift** adds an element to the start.
- **pop** takes an element from the end

The call **fruits.push(...)** is equal to **fruits[fruits.length]** = **....**

Array - Loops

One of the oldest ways to cycle array items is the for loop over indexes:

But for arrays there is another form of loop, **for..of**:

```
let fruits = ["Apple", "Orange", "Plum"];

// iterates over array elements
for (let fruit of fruits) {
  alert( fruit );
}
```

We have one more loop called **for..in**, but it's generally used on objects.

Generally, we shouldn't use for..in for arrays.

Array - Length

Another interesting thing about the length property is that it's writable.

If we increase it manually, nothing interesting happens. But if we decrease it, the array is truncated. The process is irreversible, here's the example:

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

arr.length = 2; // truncate to 2 elements
alert( arr ); // [1, 2]

arr.length = 5; // return length back
alert( arr[3] ); // undefined: the values do not return
```

So, the simplest way to clear the array is: **arr.length = 0;**

Array - Multidimensional Array

Arrays can have items that are also arrays. We can use it for multidimensional arrays, for example to store matrices:

```
let matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

alert( matrix[1][1] ); // 5, the central element
```

Array - toString method

Arrays have their own implementation of toString method that returns a comma-separated list of elements.

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

alert( arr ); // 1,2,3
alert( String(arr) === '1,2,3' ); // true
```

```
alert( [] + 1 ); // "1"
alert( [1] + 1 ); // "11"
alert( [1,2] + 1 ); // "1,21"
```

Functions

Quite often we need to perform a similar action in many places of the script.

For example, we need to show a nice-looking message when a visitor logs in, logs out and maybe somewhere else.

Functions are the main “building blocks” of the program. They allow the code to be called many times without repetition.

We’ve already seen examples of built-in functions, like alert(message), prompt(message, default) and confirm(question). But we can create functions of our own as well. Let’s see how to do that

Functions Declaration

To create a function we can use a function declaration.

The function keyword goes first, then goes the name of the function, then a list of parameters between the parentheses (comma-separated, empty in the example above, we'll see examples later) and finally the code of the function, also named "the function body", between curly braces.

```
1  function name(parameter1, parameter2, ... parameterN) {
2    // body
3  }
```

Functions Calling

```
1  function showMessage() {
2    alert( 'Hello everyone!' );
3  }
```

```
1  function showMessage() {
2    alert( 'Hello everyone!' );
3  }
4
5  showMessage();
6  showMessage();
```

The call showMessage() executes the code of the function. Here we will see the message two times.

This example clearly demonstrates one of the main purposes of functions: to avoid code duplication.

Functions - Local Variables

A variable declared inside a function is only visible inside that function.

```
1  function showMessage() {
2    let message = "Hello, I'm JavaScript!"; // local variable
3
4    alert( message );
5  }
6
7  showMessage(); // Hello, I'm JavaScript!
8
9  alert( message ); // <-- Error! The variable is local to the function
```

Functions - Outer Variables

A function can access an outer variable as well, for example: ↠

```
let userName = 'John';

function showMessage() {
  let message = 'Hello, ' + userName;
  alert(message);
}

showMessage(); // Hello, John
```

The function has full access to the outer variable. It can modify it as well. ↠

```
let userName = 'John';

function showMessage() {
  userName = "Bob"; // (1) changed the outer variable

  let message = 'Hello, ' + userName;
  alert(message);
}

alert( userName ); // John before the function call

showMessage();

alert( userName ); // Bob, the value was modified by the function
```

Functions - Outer Variables

The outer variable is only used if there's no local one.

If a same-named variable is declared inside the function then it shadows the outer one. For instance, in the code below the function uses the local userName. The outer one is ignored:

```
let userName = 'John';

function showMessage() {
  let userName = "Bob"; // declare a local variable

  let message = 'Hello, ' + userName; // Bob
  alert(message);
}

// the function will create and use its own userName
showMessage();

alert( userName ); // John, unchanged, the function did not access the outer variable
```

Functions - Global Variables

Variables declared outside of any function, such as the outer userName in the code above, are called global.

Global variables are visible from any function (unless shadowed by locals).

It’s a good practice to minimize the use of global variables. Modern code has few or no globals. Most variables reside in their functions. Sometimes though, they can be useful to store project-level data.

Functions - Parameters

We can pass arbitrary data to functions using parameters.

In the example below, the function has two parameters: **from** and **text**.

```
1  function showMessage(from, text) { // parameters: from, text
2    alert(from + ': ' + text);
3  }
4
5  showMessage('Ann', 'Hello!'); // Ann: Hello! (*)
6  showMessage('Ann', "What's up?"); // Ann: What's up? (**)
```

When the function is called in lines (5) and (6), the given values are copied to local variables **from** and **text**. Then the function uses them.

Functions - Parameters

Here's one more example: we have a variable from and pass it to the function. Please note: the function changes from, but the change is not seen outside, because a function always gets a copy of the value:

```
function showMessage(from, text) {

  from = '*' + from + '*'; // make "from" look nicer

  alert( from + ': ' + text );
}

let from = "Ann";

showMessage(from, "Hello"); // *Ann*: Hello

// the value of "from" is the same, the function modified a local copy
alert( from ); // Ann
```

Functions - Parameters

When a value is passed as a function parameter, it's also called an argument.

In other words, to put these terms straight:

- A parameter is the variable listed inside the parentheses in the function declaration (it's a declaration time term).
- An argument is the value that is passed to the function when it is called (it's a call time term).

We declare functions listing their parameters, then call them passing arguments.

Functions - Default Values

If a function is called, but an argument is not provided, then the corresponding value becomes undefined.

For instance, the aforementioned function showMessage(from, text) can be called with a single argument:

```
1  showMessage("Ann");
```

That’s not an error. Such a call would output "*Ann*: undefined". As the value for text isn’t passed, it becomes undefined.

Functions - Default Values

We can specify the so-called “default” (to use if omitted) value for a parameter in the function declaration, using =

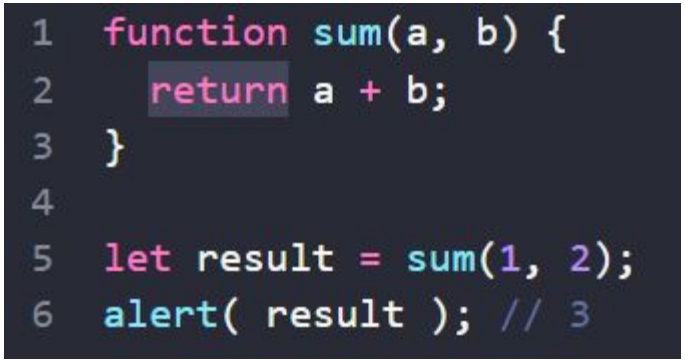
```
1  function showMessage(from, text = "no text given") {
2    alert( from + ": " + text );
3  }
4
5  showMessage("Ann"); // Ann: no text given
```

Now if the text parameter is not passed, it will get the value "no text given".

The default value also jumps in if the parameter exists, but strictly equals undefined, like this:

Functions - Returning a value

A function can return a value back into the calling code as the result.



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

let result = sum(1, 2);
alert( result ); // 3
```

The directive return can be in any place of the function. When the execution reaches it, the function stops, and the value is returned to the calling code (assigned to result above).

Functions - Returning a value

There may be many occurrences of return in a single function. For instance:

```
function checkAge(age) {
  if (age >= 18) {
    return true;
  } else {
    return confirm('Do you have permission from your parents?');
  }
}

let age = prompt('How old are you?', 18);

if ( checkAge(age) ) {
  alert( 'Access granted' );
} else {
  alert( 'Access denied' );
}
```

Functions - Returning a value

It is possible to use return without a value. That causes the function to exit immediately.

```
1  function showMovie(age) {
2    if ( !checkAge(age) ) {
3      return;
4    }
5
6    alert( "Showing you the movie" ); // (*)
7    // ...
8  }
```

In the code above, if checkAge(age) returns false, then showMovie won't proceed to the alert.

A function with an empty return or without it returns undefined

If a function does not return a value, it is the same as if it returns undefined: