JavaScript 1 - Module 2

Strings and Logic

Arrays

Objects

Functions

Solutions for JS1 Lesson 2.3 Objects

Exercise 1

Given the object:

```
let myTV = { make: "Toshiba", model: "42XV555", resolution: "1080p" };
```

a) Using bracket notation, console log out the value of the property `resolution`.

b) Console log out a string with the values of `make` and `model` concatinated (with a space between them). Use dot-notation to retrieve the two property values.

c) Add the property `year` with the value `2008` to the Object.

d) Console log out the data type of `myTV`.

e) Declear another variable, `newTV`, of the same kind of Object, with the values `LG`, `65OLEDCX`, `2160p`, `2020`.

f) Use a `for...in` loop to list all the properties of `newTV`, on the form "value (key)".

g) Add the two TV objects to an Array named `tvs`.

h) Use a `for...of` loop to list all (ie. both) Objects in the `tvs` Array, on the form: "LG 65OLEDCX (2020), 2160p".

```
// Exercise 1
console.log("Exercise 1");

let myTV = { make: "Toshiba", model: "42XV555", resolution: "1080p" };
console.log(myTV); // Just for tests

// a
console.log(myTV['resolution']); // "1080p"

// b
let makeAndModel = myTV.make + " " + myTV.model;
console.log(makeAndModel);

// c
myTV.year = 2008;
console.log(myTV); // Just for tests, again
```

```
// d
console.log(typeof myTV); // object

// e
let newTV = { make: "LG", model: "65OLEDCX", resolution: "2160p", year: 2020 };

// f
for (let prop in newTV) {
    console.log(newTV[prop] + " (" + prop + ")")
}

// g
let tvs = [myTV, newTV];
console.log(tvs); // Just for tests

// h
for (let tv of tvs) {
    console.log (tv.make + " " + tv.model + " (" + tv.year + ") " + tv.resolution);
}
```

Exercise 2 - Level 2

a) Add two more TVs to the `tvs` Array from Exercise 1, with these values, using `push()` twice:

- `TCL`, `55DP660`, `2160p`, `2018`
- `Samsung`, `QE65Q950RBT`, `4320p`, `2019`

b) Use the `Array.sort()` on `tvs` and list the result, using the same way as you did in 1h). What happens?

c) Make a *compare function* that sorts `tvs` based on the `year` value, listing the newest TVs first. Now list the sorted `tvs` Array (as in 1h and 2b).

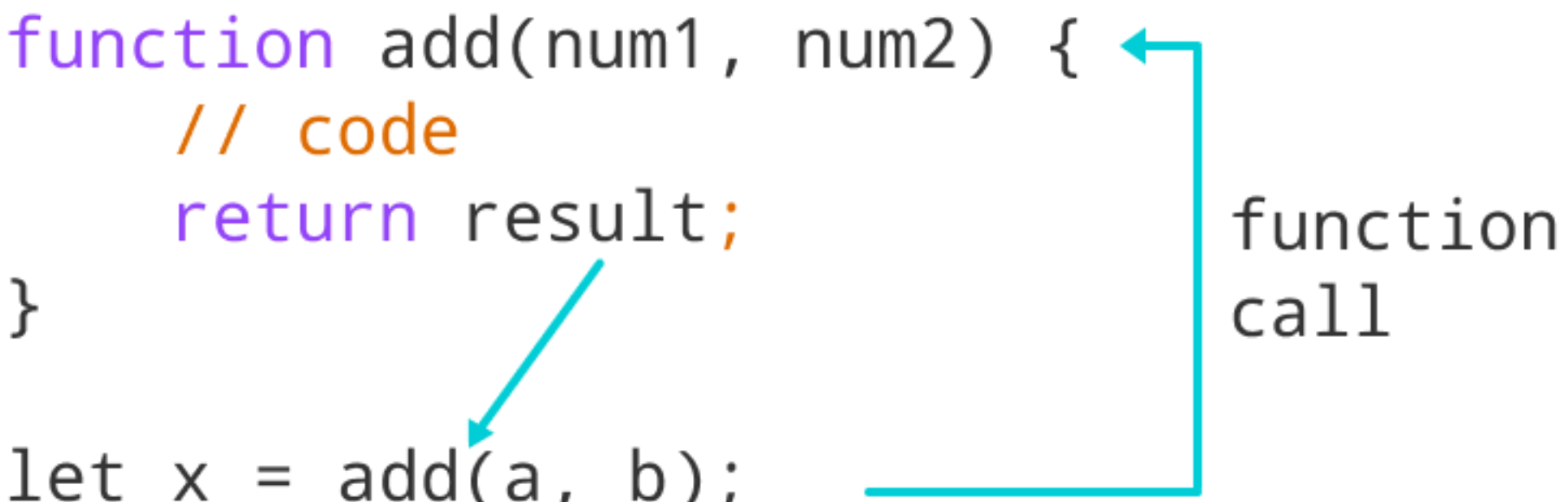
```
// Exercise 2
console.log("Exercise 2");

// a
tvs.push( { make: "TCL", model: "55DP660", resolution: "2160p", year: 2018 } );
tvs.push( { make: "Samsung", model: "QE65Q950RBT", resolution: "4320p", year: 2019 } );
console.log(tvs); // Just for tests

// b
tvs.sort();
for (let tv of tvs) {
    console.log (tv.make + " " + tv.model + " (" + tv.year + ") " + tv.resolution);
}
// Nothing happens, sort() does not know how to compare the objects

// c
tvs.sort(function(a, b){return b.year - a.year})
console.log ("Sorted list:");
for (let tv of tvs) {
    console.log (tv.make + " " + tv.model + " (" + tv.year + ") " + tv.resolution);
}
```

Introduction to functions



Source

JavaScript Functions

A JavaScript function is a block of code designed to perform a particular task.

JavaScript functions are defined with the function keyword.

You can use a function declaration or a function expression.

Syntax for declaring a **function**:

```
function fname(parameters) {
  // code to be executed
}
```

JavaScript Function Invocation

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are **invoked**.

```
function fname(parameters) {
  // code to be executed
}

fname(arguments); // function is being invoked
```

> It is common to use the term "*call a function*" instead of "*invoke a function*", or "*call upon a function*", "*start a function*", or "*execute a function*".

Function Parameters and Arguments

Function **parameters** are listed inside the parentheses `( )` in the function definition.

Function **arguments** are the values received by the function when it is invoked.

Inside the function, the arguments (now, the parameters) behave as **local variables**.

```
function fname(parameter1, parameter2) {
  // code to be executed, with access to the parameters
  let whatever = parameter1 + parameter2;
  // more code to be executed...
}

fname(argument1, argument2); // function is being invoked
```

Function **parameters** are the **names** listed in the function definition.

Function **arguments** are the real **values** passed to (and received by) the function.

Parameter Rules

- JavaScript function definitions do not specify **data types** for parameters.
- JavaScript functions do not perform **type checking** on the passed arguments.
- JavaScript functions do not check the **number of arguments** received.

Basic Function Example

```
function myOwnLog(text) {
   console.log ("I need to check this: " + text);
}

myOwnLog("whatever is this?"); // I need to check this: whatever is this?
myOwnLog("whatever is that?"); //  need to check this: whatever is that?
myOwnLog(13 + 14 + 15);        // I need to check this: 42
```

Here the function name is `myOwnLog`.

`myOwnLog` has one parameter, `text`, that receives one argument when invoked.

`myOwnLog` then console logs out the value of the argument along with some clarifying text.

> Notice: the argument doesn't have to be a string.

Function Return

When JavaScript reaches a return statement, the function will stop executing.

If the function was invoked from a statement, JavaScript will "return" to execute the code after the invoking statement.

Functions often compute a return value. The return value is "returned" back to the "caller":

```
function fname(parameter1, parameter2) {
  // code to be executed
  return someValue;
}

let newValue = fname(argument1, argument2);
// someValue is being returned from the function
// and assigned to the variable newValue
```

Function Example with return

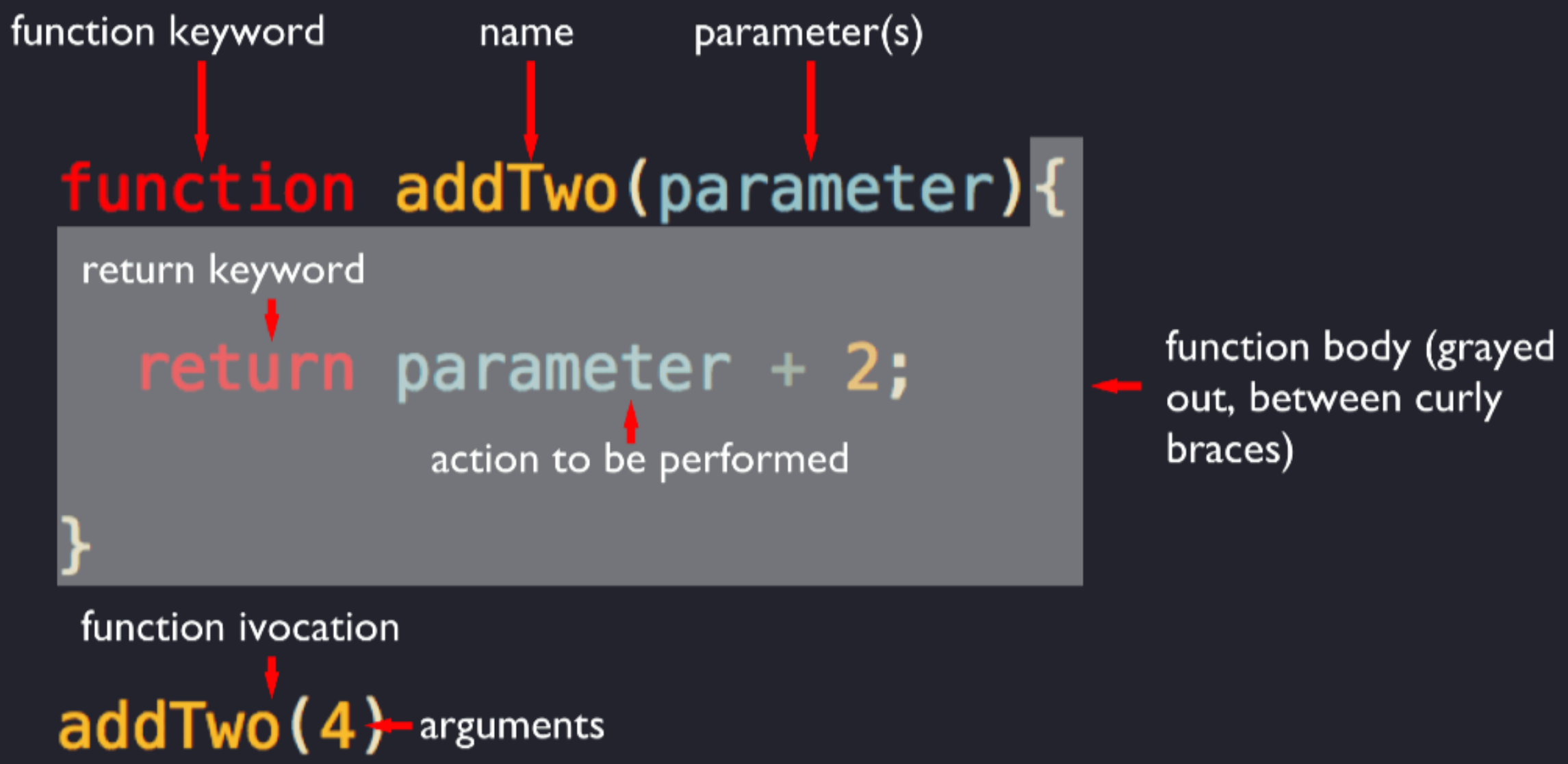
```
function myFunction(a, b) {
  return a * b;
}
console.log ( myFunction(6, 7) );
```

Here the name of the function is `myFunction`.

`myFunction(a, b)` has two parameters, `a` and `b`.

`myFunction` calculates the value of `a * b` and returns the result back to the statement that invoked it.

Here we called `myFunction` with 2 arguments, `6` and `7`, and thus the return value was `42`, which is then console logged out.



Source Function will output 6

Why Functions?

You can reuse code: Define the code once, and use it many times.

You can use the same code many times with different arguments, to produce different results.

```
// Function that calculates the distance from
// the point, given by its coordinates (x, y)
// to origo (0, 0) in a 2D coordinate system
function distanceFromOrigo(x, y) {
    let d = Math.sqrt (x**2 + y**2);
    return d;
}
let a = distanceFromOrigo(1, 1);
let b = distanceFromOrigo(3, 4);
let c = distanceFromOrigo(-2, 2);
let d = distanceFromOrigo(0, -5);
console.log (a, b, c, d); // 1.4142... 5 2.8284... 5
```

Let's extend the `distanceFromOrigo()` function to calculate the distance from a point to any other point, and not just origo.

A point can be described as an object:

```
let point = { x: 3, y: 4 };
```

First we make the new `distanceFromOrigo()` take a point object as an argument:

```
function distanceFromOrigo(point) {
    let d = Math.sqrt (point.x**2 + point.y**2);
    return d;
}
let pointA = { x: 3, y: 4 }
let distance = distanceFromOrigo(pointA);
console.log (distance); // 5
```

Then, after a renaming (since it no longer just uses origo), give it two parameters, p1 and p2 for the two points that should be calculated:

```
function distance(p1, p2) {
    let d = Math.sqrt( (p2.x - p1.x)**2 + (p2.y - p1.y)**2 );
    return d;
}
let pointA = { x: 3, y: 4 }
let pointB = { x: 0, y: 0 }
let pointC = { x: -1, y: -1 }

let a = distance(pointA, pointB);
let b = distance(pointB, pointC);
let c = distance(pointA, pointC);
console.log (a, b, c); // 5 1.41... 6.40...
```

> Note: The Formula for 2D Euclidean Distance, ie. the straight line distance between points in two dimensions, used in all these examples came from this page.

Make a function to list values from Object

Given the Objects (from the Lesson 2.3 exercises):

```
let tv1 = { make: "Toshiba", model: "42XV555", resolution: "1080p", year: 2008 };
let tv2 = { make: "LG", model: "65OLEDCX", resolution: "2160p", year: 2020 };
let tv3 = { make: "TCL", model: "55DP660", resolution: "2160p", year: 2018 };
let tv4 = { make: "Samsung", model: "QE65Q950RBT", resolution: "4320p", year: 2019 };
```

We want to make a function to print the values of the TVs properties on the form: "make model (year), resolution", eg. "Toshiba 42XV555 (2008), 1080p".

```
function describeTV (tv) {
    return `${tv.make} ${tv.model} (${tv.year}), ${tv.resolution}`;
}
console.log (describeTV(tv1)); // Toshiba 42XV555 (2008), 1080p
console.log (describeTV(tv3)); // TCL 55DP660 (2018), 2160p
```

We can add the TVs to an array and loop over the array, to list them, using the new function:

```
var tvs = [tv1, tv2, tv3, tv4];
for (let tv of tvs) {
    console.log (describeTV(tv));
}
```

To get the output:

```
Toshiba 42XV555 (2008), 1080p
LG 65OLEDCX (2020), 2160p
TCL 55DP660 (2018), 2160p
Samsung QE65Q950RBT (2019), 4320p
```

> Note: Now you can use this to improve the exercise from Lesson 2.3.

Default function parameters

In JavaScript, function parameters default to `undefined`. However, it's often useful to set a different **default value**. This is where default parameters can help.

In the past, the general strategy for setting defaults was to test parameter values in the function body and assign a value if they are undefined.

In the following example, if no value is provided for `b` when multiply is called, `b`'s value would be undefined when evaluating `a * b` and multiply would return `NaN`:

```
function multiply(a, b) {
  return a * b;
}

console.log (multiply(5, 2));  // 10
console.log (multiply(5));     // NaN !
```

To guard against this, something like the second line would be used, where `b` is set to `1` if multiply is called with only one argument:

```
function multiply(a, b) {
  b = (typeof b !== 'undefined') ? b : 1;
  return a * b;
}

console.log (multiply(5, 2));  // 10
console.log (multiply(5));     // 5
```

Default function parameters. cont.

Default function parameters allow named parameters to be initialized with default values if no value or undefined is passed.

```
function multiply(a, b = 1) {
  return a * b;
}

console.log(multiply(5, 2)); // expected output: 10
console.log(multiply(5)); // expected output: 5
console.log(multiply(5, undefined)); // expected output: 5
```

So, with default parameters in ES2015, checks in the function body are no longer necessary.

Arrow Functions

Given this regular function-declaration:

```
function add(x, y) {
  return x + y;
}
```

This could also be acheived with a Function Expression in ES5:

```
// ES5 function expression
var add = function(x, y) {
  return x + y;
}
```

Arrow functions in ES6 allows a short syntax when writing **function expressions**:

```
// ES6 function expression
const add = (x, y) => x + y;
```

You don't need the function keyword, the return keyword, nor the curly brackets.

Arrow functions do not have their own `this`. They are not well suited for defining object methods.

Arrow functions are not hoisted. They must be defined before they are used.

Using `const` is safer than using `var`, because a function expression is always constant value.

You can *only* omit the `return` keyword and the curly brackets, `{}`, if the function is a single statement. Because of this, it might be a good habit to always keep them:

```
const add = (x, y) => { return x + y };

let a = add(13, 29);
let b = add(5, 8);
let c = add(a, b);
console.log(a, b, c); // 42 - 13 - 55
```

JavaScript Events

HTML events are "things" that happen to HTML elements.

When JavaScript is used in HTML pages, JavaScript can "react" on these events, eg. when a button was clicked:

```
<button onclick="displayDate()">The time is?</button>
<p id="demo"></p>
```

```
function displayDate() {
    const target = document.getElementById("demo");
    target.innerText = Date();
}
```

Something like "Fri Jan 14 2022 11:03:37 GMT+0100 (CET)" will be written out in the `p#demo` element.

Codepen

Another example: When the button is pressed, the `changeStyle()` function is called, with the argument which is the selector for the target element, `"p#demo"`.

```
<button onclick='changeStyle("p#demo")'>Push me</button>
<p id="demo">Lorem ipsum dolor sit amet</p>
```

The function, changes the style of the target element:

```
function changeStyle(targetElement) {
    const target = document.querySelector(targetElement);
    target.style.color = "blue";
    target.style.backgroundColor = "beige";
    target.style.fontSize = "24px";
}
```

> Reading tip: querySelector vs. getElementById: A Comparison

Codepen

And another example

```
<ul id="myList"></ul>
<button onclick="list()">List elements</button>
<button onclick="sort()">Sort elements</button>
```

Here we make two functions, one that lists the content of an Array, in the original order, and another that sorts is, then lists it:

```
const out = document.querySelector("ul#myList"); // Target element
var myList = ["vg.no", "dagbladet.no", "nrk.no", "bt.no", "ba.no", "klassekampen.no"];

function list() {
    for (let item of myList) {
        out.innerHTML += "<li>" + item + "</li>";
    }
}
function sort() {
    myList.sort();
    for (let item of myList) {
        out.innerHTML += "<li>" + item + "</li>";
    }
}
```

Rooms for improvement

Note that the code above has some serious issues:

1. Each time you (re-) click any of the buttons, new, duplicated items are added to the list.
2. When `myList` is sorted, the original order of the Array is changed, and the Array is forever sorted.
3. We see some hints of "Copy-and-paste programming", since the two functions both have identical `for` loops.

We can solve these issues...

1. Empty the HTML list element before listing the Array again:

```
const out = document.querySelector("ul#myList"); // Target element
let myList = ["vg.no", "dagbladet.no", "nrk.no", "bt.no", "ba.no", "klassekampen.no"];
function list() {
    out.innerHTML = ""; // Empty the list before listing
    for (let item of myList) {
        out.innerHTML += "<li>" + item + "</li>";
    }
}
function sort() {
    out.innerHTML = ""; // Empty the list before listing
    myList.sort();
    for (let item of myList) {
        out.innerHTML += "<li>" + item + "</li>";
    }
}
```

2. Use `slice()`, to copy the original Array, then sort the copy and list out the copied Array:

```
const out = document.querySelector("ul#myList"); // Target element
let myList = ["vg.no", "dagbladet.no", "nrk.no", "bt.no", "ba.no", "klassekampen.no"];
function list() {
    out.innerHTML = ""; // Empty the list before listing
    for (let item of myList) {
        out.innerHTML += "<li>" + item + "</li>";
    }
}
function sort() {
    out.innerHTML = ""; // Empty the list before listing
    // Use myList.slice() to make a copy of myList,
    // then sort that immediately and
    // assign the new Array to sortedList
    let sortedList = myList.slice().sort();
    for (let item of sortedList) { // List out the sortedList
        out.innerHTML += "<li>" + item + "</li>";
    }
}
```

3. Make a helper function, `listArray(array, element)`, that takes two parameters: The Array it should list and the HTML element it should list it to:

```
const out = document.querySelector("ul#myList"); // Target element
let myList = ["vg.no", "dagbladet.no", "nrk.no", "bt.no", "ba.no", "klassekampen.no"];

function list() {
    listArray(myList, out);
}
function sort() {
    // Use myList.slice() to make a copy of myList,
    // then sort that immediately and assign the new Array to sortedList
    let sortedList = myList.slice().sort();
    listArray(sortedList, out);
}
function listArray(array, element) {
    element.innerHTML = ""; // Empty the list before listing
    for (let item of array) {
        element.innerHTML += "<li>" + item + "</li>";
    }
}
```

Codepen, final verision

Finally, a small warning about parameters/arguments

Tecnically, if you pass an argument into a function, then you're sending the value of that argument (variable or literal) into the function.

That means the original variable is not affected of whatever goes on in the function:

```
var a = 2;

function doubleMe(x) {
    x *= 2;
    return x;
}

console.log (a); // 2
console.log (doubleMe(a)); // 4
console.log (a); // 2, the original value is unchanged
```

However, if an *object* is used as an argument, then the value isn't sent in, but only a reference (often shown with a leadning `&`) to the original object.

This means that changes to the object inside the function, also changes the original:

```
var a = { dill: 2, dall: 4 };

function doubleMe(x) {
  x.dill *= 2;
  x.dall *= 2;
  return x;
}

console.log (a); // { 2, 4 }
console.log (doubleMe(a)); // { 4, 8 }
console.log (a); // { 4, 8 }, the original value has been changed
```

Sources and resources

JavaScript Function Definitions

JavaScript Function Parameters

JavaScript Function Invocation

JavaScript building blocks > Build your own function

JavaScript reference > Functions

Bonus: **JavaScript Best Practices**

Bonus: **JavaScript Common Mistakes**

Bonus: **What went wrong? Troubleshooting JavaScript**

Todos

Github Classroom

JS1 Lesson 2.4 Functions

Mollify

Read Introduction to Functions, and do the Lesson Task.