JavaScript everyday commands

Index

Conversions String <-> Number

String to Number
let str = “12345”;

Example:	Result:

let n1 = Number(str)
let n2 = +str;
 	
console.log(n1);
console.log(n2);
	
	12345
	12345

Number to String
let n = 12345

Example:	Result:

let str1 = String(n)
let str2 = “ ” + n;
 	
console.log(str1);
console.log(str2);
	
	“12345”
	“12345”

String

Variable used for the examples: const msg = “It’s a beautiful day”
(Without relying on the previous examples. Each time a new “It’s a beautiful day“ string)

.length

Returns the length of the string. Warning: Some special characters return two as lengths.

Example:	Result:
let n = msg.length;	n = 20

.toUpperCase()

Converts a string to uppercase letters.

Example:	Result:
msg.toUpperCase();	“IT’S A BEAUTIFUL DAY”

.toLowerCase()

Converts a string to lowercase letters.

Example:	Result:
msg.toLowerCase();	“it’s a beautiful day”

.trim()

Remove all white spaces in the beginning and end of a string.

Example:	Result:
const example = “ It’s a beautiful day ” example.trim();	“It’s a beautiful day”

.indexOf()

Returns position of a search value. Starting position is optional.

> str.indexOf(searchValue, startingPosition);

Example:	Result:
msg.indexOf(“u”,11);	14 // finds the second “u” in “beautiful”

.substring()

Returns a selected part of the string.

> str.substring(start,end);

Example:	Result:
msg.substring(4,8);	“a beaut”

.split

Split a string to an array, based on the separator. If no separator is defined, it’s splitted by each character. The limit is how many items should be added to the array. The items after the limit will not be considered.

> string.split(*separator*, *limit*)

Example:	Result:
let str = “P-e-t-e-r” let arr = str.split(”-”); let arr2 = str.split(”-”, 3);	 [”P”,”e”,”t”,”e”,”r”] [”P”,”e”,”t”]

Boolean() and if statements results

The following values result in false with a Boolean() function or if() statement.

Table 11 — ToBoolean Conversions

Argument Type	*Result*
Undefined	**false**
Null	**false**
Boolean	The result equals the input argument (no conversion).
Number	The result is **false** if the argument is **+0**, **–0**, or **NaN**; otherwise the result is **true**.
String	The result is **false** if the argument is the empty String (its length is zero); otherwise the result is **true**.
Object	**true**

Example:	Result:

if(-0){
console.log(“True”);
}else{
console.log(“false”);
}
	“false” // because 0, -0 or NaN is false

Array

Variable used for the examples: const guys = [“Harry” , “Snape” , “Ron”]
(Without relying on the previous examples. each time a new [“Harry”,”Snape”,”Ron”] array)

.push

Add new item at the end of the array.

Example:	Result:
guys.push(“Ted”);	[“Harry”,”Snape”,”Ron”,”Ted”]

.pop

Remove the last item of the array.

Example:	Result:
guys.pop();	[“Harry”,”Snape”]

.unshift

Add new item to the beginning of the array.

Example:	Result:
guys.unshift(“Ted”);	[“Ted”,“Harry”,”Snape”,”Ron”]

.shift

Remove first item of the array.

Example:	Result:
guys.shift();	[”Snape”,”Ron”]

.slice

Return only a part of the array. From … to without modifying the array.
> arr.slice(begin, end)

Example:	Result:
guys.slice(1,2);	[”Snape”,”Ron”] //but harry still exists

.splice

Insert new items to the array at a specific position and/or remove items at a specific position in the array.
> array.splice(index, deleteCount, item1, item2, item3,…)

Example:	Result:
guys.splice(1,1,”Billy”,”Hilly”);	[”Harry”,”Billy”,”Hilly”,”Ron”]

.join

Return all items of an array as **a string**, separated with whatever character you decide.
> array.join(seperator)

Example:	Result:
guys.join(“-”);	”Harry-Billy-Hilly-Ron”

Array.from

Convert a string to an array

Example:	Result:
let str = “Peter” let arr = Array.from(str);	[”P”,”e”,”t”,”e”,”r”]

.split

Split a string to an array, based on the separator. If no separator is defined, it's splitted by each character. The limit is how many items should be added to the array. The items after the limit will not be considered.

> string.split(*separator*, *limit*)

Example:	Result:

let str = "P-e-t-e-r"
let arr = str.split("-");
let arr2 = str.split("-", 3);
	
	["P","e","t","e","r"]
	["P","e","t"]

.sort(function)

Sort's an array based on a function that returns either 1, -1 or 0.

> array.sort(function)

If the function returns:

- **1** -> then move the array item to the right
- **-1** -> then move the array item to the left
- **0** -> then keep the array items current position

Sorting based on numbers

Example:

```
const numb = [2, 7, 9, 3, 1];

function compareNumber(a, b) {
  if (a > b) {
    return 1;
  } else {
    return -1;
  }
}

numb.sort(compareNumber);

console.log(numb);
```

Result:

[1,2,3,7,9]

Explanation of previous code:

1. 2 > 7 returns false, return -1 and move 2 to the left -> [2,7,9,3,1]
2. 7 > 9 returns false, return -1 and move 7 to the left -> [2,7,9,3,1]
3. 9 > 3 returns true, return 1 and move 9 to the right -> [2,7,3,9,1]
4. 9 > 1 returns true, return 1 and move 9 to the right -> [2,7,3,1,9]

Now 9 is at the very end. .sort() runs in a loop so many time until every number has been sorted into it's correct position.

Sorting based on characters

Example:	Result:

`const names = ["Cindy", "Alberto", "Peter", "Marcus"];`
 	
`function compareNames(a, b) {`
`  if (a > b) {`
`  return 1;`
`  } else {`
`  return -1;`
`  }`
`}`
 	
`names.sort(compareNames);`
`console.log(names);`
	[“Alberto”,”Cindy”,”Marcus,”Peter”]

Using strings, the Boolean results are decided based on the first letter (if not otherwhise defined) and its “computer generated” decimal position.
(Have a look at the table below)

Decimal	Character
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z

Decimal	Character
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z

.filter(function)

Returns a new array with all the elements which succeeded a specific test. filter uses a function that either returns true (= add it to the new array) or false(= don't add it to the new array).

Example:	Result:

`const age = [21, 28, 11, 17, 88, 13];`
 	
`const passedAge = age.filter(ageTest);`
 	
`function ageTest(a) {`
`  if (a > 18) {`
`  return true;`
`  } else {`
`  return false;`
`  }`
`}`
 	
`console.log(passedAge);`
	[21,28,88]

Shorter version of ageTest, they are both the same
`const age = [21, 28, 11, 17, 88, 13];`
 	
`const passedAge = age.filter(ageTest);`
 	
`function ageTest(a) {`
`  return a > 18`
`}`
 	
`console.log(passedAge);`
	[21,28,88]

Objects

Variable used for the examples:
```
const student = {
    firstName : "Harry",
    lastName : "Potter",
    "parents" : "Dead"
}
```
(Without relying on the previous examples. Each time a new student object with those three properties)
firstName, lastName and “parents” are called **property keys**.
The relative values are called **property values**.

Accessing a property

To access a properties value, we use the dot.notation to navigate deeper into the object tree.

Example:	Result:
console.log(student.lastName)	”Potter”

Some property keys can not be written with the dot.notation because they are either not forbidden property key names or variables. They have to be escaped with brackets.
To read more -> Computed property names

Example:	Result:
console.log(student.parents) console.log(student[parents])	-> returns error because parents is a string "Dead"

Modify objects directly

To modify objects directly we either modify an existing property or create a new one.

Example:	Result:

student.lastName = "Wizard";
console.log(student);
// Existing property
	{
	firstName : "Harry",
	lastName : "Wizard",
	"parents" : "Dead"
	}

student.house = "Gryfindor";
console.log(student);
// Create a new property
	{
	firstName : "Harry",
	house : "Gryfindor",
	lastName : "Potter",
	"parents" : "Dead"
	}

Object methods

There are two ways to add function / methods to an object.
Either by referring to an existing function or by adding a new method within the object.

Example:	Result:

`function ourtoString(){`
`  return student.firstName + " " + student.lastName;`
`}`
`student.toString = ourToString();`
`console.log("The student is" + student.toString);`
`// Existing function`
	"The student is Harry Potter"

`const student = {`
`  firstName : "Harry",`
`  lastName : "Potter",`
`  "parents" : "Dead",`
`  toString : function(){`
`  return this.firstName + " " + this.lastName;`
`  }`
`}`
`console.log("The student is" + student.toString);`
`// Adding method to object first way`
	"The student is Harry Potter"

`const student = {`
`  firstName : "Harry",`
`  lastName : "Potter",`
`  "parents" : "Dead",`
`  toString() {`
`  return this.firstName + " " + this.lastName;`
`  }`
`}`
`console.log("The student is" + student.toString);`
`// Adding method to object second way`
`// Both ways are valid and the same`
	"The student is Harry Potter"

Warning: Defining methods within objects only works with objects and **not JSON!**

.toString()

By default, each object has a .toString() method (even if we have not defined it).
The .toString() method is called whenever we try to convert an object to a string.
E.g.

```
const student = {
  firstName: "Harry",
  lastName: "Potter"
};

// Three ways to convert object to string
console.log("The name " + String(student)); //returns "The name [object Object]"
console.log("The name " + student); //returns "The name [object Object]"
console.log("The name " + student.toString()); //returns "The name [object Object]"
```

You can create a function to be called in place of the default toString() method. The toString() method takes no arguments and should return a string.
E.g.

```
const student = {
  firstName: "Harry",
  lastName: "Potter",
  toString() {
    return this.firstName + " " + this.lastName;
  }
};

// Three ways to convert object to string
console.log("The name " + String(student)); //returns "The name Harry Potter"
console.log("The name " + student); //returns "The name Harry Potter"
console.log("The name " + student.toString()); //returns "The name Harry Potter"
```

Prototypes

Having many objects, this would be a bad habit to create them:

```
const student1 = {
    firstName: "Harry",
    lastName: "Potter",
    toString() {
        return this.firstName + " " + this.lastName;
    }
}

const student2 = {
    firstName: "Ron",
    lastName: "Weasley",
    toString() {
        return this.firstName + " " + this.lastName;
    }
}

const student3 = {
    firstName: "Hermione",
    lastName: "Granger",
    toString() {
        return this.firstName + " " + this.lastName;
    }
}

const student4 = {
    firstName: "Neville",
    lastName: "Longbottom",
    toString() {
        return this.firstName + " " + this.lastName;
    }
}
```

That's why we can create prototypes of objects where we define the structure, create new objects based on this structure and populate them with the missing data.

Using .create()

Using a prototype, we can use the .create() method to create new objects.

```
const Student = {
  firstName : "",
  lastName : "",
  toString() {
    return this.firstName + " " + this.lastName;
  }
}

const student1 = Object.create(Student);
const student2 = Object.create(Student);

student1.firstName = "Harry";
student2.lastName = "Potter";

student2.firstName = "Ron";
student2.lastName = "Weasley";
```

It is considered good practice to name a prototype with an upper-case first letter.

Using a constructor function

Note: This has not been thaught in the class, only for educational purposes have I added this and it's not needed.

A constructor is a function, that takes arguments and creates an object based on these passed arguments.

```
function Student(first, last){ //constructor named Student
  this.firstName = first;
  this.lastName = last;
  this.toString = function(){
    return this.firstName + " " + this.lastName;
  }
}

// Create objects based on structure
const student1 = new Student("Harry", "Potter");
const student2 = new Student("Ron", "Weasley");
```

However, for performance issues, it's recommended to create methods using the **.prototype** property.

```
function Student(first, last) {
  //constructor named Student
  this.firstName = first;
  this.lastName = last;
}

//Using the prototype property
Student.prototype.toString = function() {
  return this.firstName + " " + this.lastName;
}

// Create objects based on structure
const student1 = new Student("Harry", "Potter");
const student2 = new Student("Ron", "Weasley");
```

If you want to read more about the constructor ->
https://www.w3schools.com/js/js_object_constructors.asp
and
https://www.w3schools.com/js/js_object_prototypes.asp

Object.keys(object)

Returns every top-level property key of an object as an array.

Example:	Result:
Object.keys(student)	["firstName", "lastName", "parents"]