My weekly study!! (week03)

I tested all the code above !!!!!

Learning bout “this:”

**To access the object, a method can use the “this” keyword.**

let user = {

name: "John",

age: 30,

sayHi() {

// "this" is the "current object"

alert(this.name);

}

};

user.sayHi(); // John

Now, pay Attention on it:

Using the “user” instead of “this”

let user = {

name: "John",

age: 30,

sayHi() {

alert(user.name); // "user" instead of "this"

}

};

Results: Using “this.name” the code will work!!!

let user = {

name: "John",

age: 30,

sayHi() {

alert( user.name ); // leads to an error

}

};

let admin = user;

user = null; // overwrite to make things obvious

admin.sayHi(); // Whoops! inside sayHi(), the old name is used! error!

Now: Pay attention:

let user = {

name: "John",

hi() { alert(this.name); }

}

// split getting and calling the method in two lines

let hi = user.hi;

hi(); // Error, because this is undefined

**To make user.hi() calls work, JavaScript uses a trick – the dot '.' returns not a function, but a value of the special**[**Reference Type**](https://tc39.github.io/ecma262/#sec-reference-specification-type)**.**

The Reference Type is a “specification type”. We can’t explicitly use it, but it is used internally by the language.

The value of Reference Type is a three-value combination (base, name, strict), where:

* base is the object.
* name is the property name.
* strict is true if use strict is in effect.

The result of a property access user.hi is not a function, but a value of Reference Type. For user.hi in strict mode it is:

// Reference Type value

(user, "hi", true)

**I like to learn about that:**

[**Arrow functions have no “this”**](https://javascript.info/object-methods#arrow-functions-have-no-this)

Arrow functions are special: they don’t have their “own” this. If we reference this from such a function, it’s taken from the outer “normal” function.

For instance, here arrow() uses this from the outer user.sayHi() method:

let user = {

firstName: "Ilya",

sayHi() {

let arrow = () => alert(this.firstName);

arrow();

}

};

user.sayHi(); // Ilya

### 

### [Explain the value of "this"](https://javascript.info/object-methods#explain-the-value-of-this) :

In the code below we intend to call user.go() method 4 times in a row.

But calls (1) and (2) works differently from (3) and (4). Why?

let obj, method;

obj = {

go: function() { alert(this); }

};

obj.go(); // (1) [object Object]

(obj.go)(); // (2) [object Object]

(method = obj.go)(); // (3) undefined

(obj.go || obj.stop)(); // (4) undefined

Great explanation!!!

1. That’s a regular object method call.
2. The same, brackets do not change the order of operations here, the dot is first anyway.
3. Here we have a more complex call (expression).method(). The call works as if it were split into two lines:
4. f = obj.go; // calculate the expression

f(); // call what we have

Here f() is executed as a function, without this.

1. The similar thing as (3), to the left of the dot . we have an expression.

To explain the behavior of (3) and (4) we need to recall that property accessors (dot or square brackets) return a value of the Reference Type.

Any operation on it except a method call (like assignment = or ||) turns it into an ordinary value, which does not carry the information allowing to set this.

### [Create a calculator](https://javascript.info/object-methods" \l "create-a-calculator)

importance: 5

Create an object calculator with three methods:

* read() prompts for two values and saves them as object properties.
* sum() returns the sum of saved values.
* mul() multiplies saved values and returns the result.

let calculator = {

// ... your code ...

};

calculator.read();

alert( calculator.sum() );

alert( calculator.mul() );

let calculator = {

sum() {

return this.a + this.b;

},

mul() {

return this.a \* this.b;

},

read() {

this.a = +prompt('a?', 0);

this.b = +prompt('b?', 0);

}

};

calculator.read();

alert( calculator.sum() );

alert( calculator.mul() );

### **Looping Over Maps**

Maps are also enumerable, so it's also possible to loop over a map in a similar way to a set. The loop will iterate over each key-value pair in the same order as they were added to the map. For example let's use the romanNumerals map that we created earlier:

const romanNumerals = new Map();

romanNumerals.set(1,'I').set(2,'II').set(3,'III').set(4,'IV').set(5,'V');

romanNumerals

<< Map { 1 => 'I', 2 => 'II', 3 => 'III', 4 => 'IV', 5 => 'V' }

Every map object has a keys() method lets us iterate over each key with the following for-of loop:

for(const key of romanNumerals.keys()) {

console.log(key);

}

<< 1

2

3

4

5

There is also a values() method that lets us iterate over the values in a similar way:

for(const value of RomanNumerals.values()) {

console.log(value);

}

<< I

II

III

IV

V

Objects!!!!

Creating:

To create an object literal, simply enter a pair of curly braces. The following example creates an empty object that is assigned to the variable spiderman :

const spiderman = {};

It’s also possible to create an object using a constructor function. This example will also create an empty object:

const spiderman = new Object();

### **Accessing Properties**

You can access the properties of an object using the dot notation that we’ve already seen in previous chapters. This will return the value of that property, as can be seen in the example below:

superman.name

<< 'Superman'

You can also access an object’s properties using bracket notation ― the property is represented by a string inside square brackets, so needs to be placed inside single or double quotation marks:

superman['name']

<< 'Superman'

superman["real" + " " + "name"] // the property is built using string concatenation

<< "Clark Kent"

## Calling Methods

To call an object’s method we can also use dot or bracket notation. Calling a method is the same as invoking a function, so parentheses need to be placed after the method name:

superman.fly()

<< 'Up, up and away!'

superman['fly']()

<< 'Up, up and away!'

### **Adding Properties**

New properties and methods can be added to objects at any time in a program. This is done by simply assigning a value to the new property. For example, if we wanted to add a new city property to our superman object, we would do it like so:

superman.city = 'Metropolis';

<< 'Metropolis'

Now if we take a look at the superman object, we can see that it has a city property:

superman

<< { name: 'Superman',

'real name': 'Clark Kent',

height: 75,

weight: 235,

hero: true,

villain: false,

allies: [ 'Batman', 'Supergirl', 'Superboy' ],

fly: [Function: fly]

city: 'Metropolis' }

## Finding all the Properties of an Object

We can loop through all of an object’s properties and methods by using a for in loop. For example, to log all the properties of the superman object to the console, we could use:

for(const key in superman) {

console.log(key + ": " + superman[key]);

}

<< "name: Superman"

<< "real name: Clark Kent"

<< "height: 75"

<< "weight: 235"

<< "hero: true"

<< "villain: false"

<< "allies: Batman,Supergirl,Superboy"

<< "fly: function (){

console.log(\"Up, up and away!\");

}"

for(const key in superman) {

if(superman.hasOwnProperty(key)){

console.log(key + ": " + superman[key]);

}

}

MATH Object:

#### **Rounding Methods**

The Math.ceil() method will round a numberupto the next integer, or remain the same if it is already an integer:

Math.ceil(4.2);

<< 5

Math.ceil(8);

<< 8

Math.ceil(-4.2);

<< -4

he Math.floor() method will round a number*down*to the next integer, or remain the same if it is already an integer:

Math.floor(4.2);

<< 4

Math.floor(8);

<< 8

Math.floor(-4.2);

<< -5

The Math.round() method will round a number to the*nearest*integer:

Math.round(4.5);

<< 5

Math.round(4.499);

<< 4

Math.round(-4.2);

<< -4

ES6 also introduced the Math.trunc() method that returns the integer-part of a number – that is, it gets truncated at the decimal point:

Math.trunc(4.9);

<< 4

Math.trunc(-4.2);

<< -4

#### **Maximum & Minimum Methods**

The Math.max() method returns the maximum number from its arguments:

Math.max(1,2,3);

<< 3

Math.max(Math.PI,Math.SQRT2, Math.E);

<< 3.141592653589793

And the Math.min() method unsurprisingly returns the minimum number from the given arguments:

Math.min(1,2,3);

<< 1

Math.min(Math.PI,Math.SQRT2, Math.E);

<< 1.4142135623730951