June 19, 2022

Objects

As we know from the chapter Data types, there are eight data types in JavaScript. Seven of them are called “primitive”, because their values contain only a single thing (be it a string or a number or whatever).

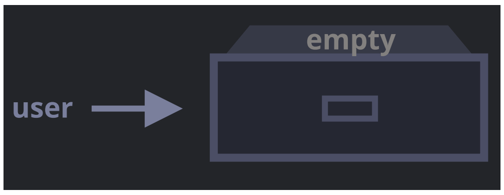
In contrast, objects are used to store keyed collections of various data and more complex entities. In JavaScript, objects penetrate almost every aspect of the language. So we must understand them first before going in-depth anywhere else.

An object can be created with figure brackets `{…}` with an optional list of *properties*. A property is a “key: value” pair, where `key` is a string (also called a “property name”), and `value` can be anything.

We can imagine an object as a cabinet with signed files. Every piece of data is stored in its file by the key. It’s easy to find a file by its name or add/remove a file.

An empty object (“empty cabinet”) can be created using one of two syntaxes:

```
let user = new Object(); // "object constructor" syntax
let user = {};  // "object literal" syntax
```



Usually, the figure brackets `{...}` are used. That declaration is called an *object literal*.

Literals and properties

We can immediately put some properties into `{...}` as “key: value” pairs:

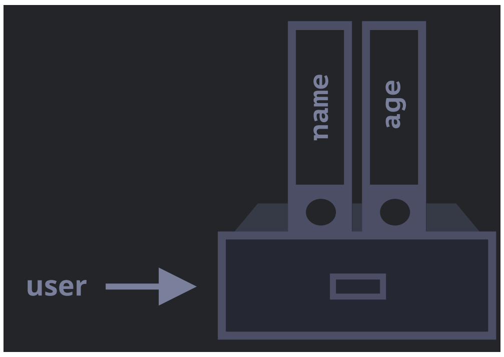
```
let user = {     // an object
  name: "John",  // by key "name" store value "John"
  age: 30        // by key "age" store value 30
};
```

A property has a key (also known as “name” or “identifier”) before the colon `":"` and a value to the right of it.

In the `user` object, there are two properties:

1. The first property has the name `"name"` and the value `"John"`.
2. The second one has the name `"age"` and the value `30`.

The resulting `user` object can be imagined as a cabinet with two signed files labeled “name” and “age”.



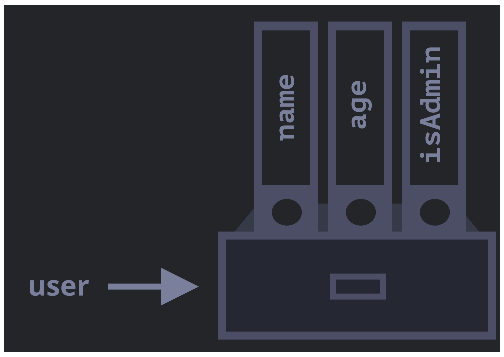
We can add, remove and read files from it at any time.

Property values are accessible using the dot notation:

```
// get property values of the object:
alert( user.name ); // John
alert( user.age ); // 30
```

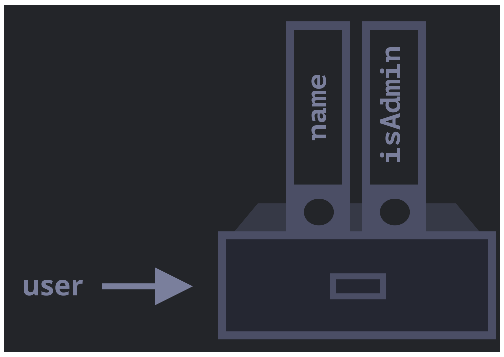
The value can be of any type. Let’s add a boolean one:

```
user.isAdmin = true;
```



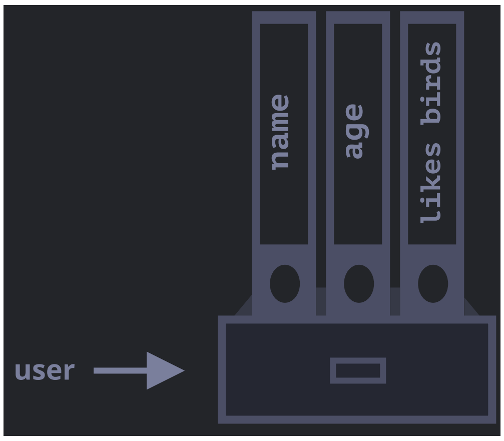
To remove a property, we can use the `delete` operator:

```
delete user.age;
```



We can also use multiword property names, but then they must be quoted:

```
let user = {
  name: "John",
  age: 30,
  "likes birds": true  // multiword property name must be quoted
};
```



The last property in the list may end with a comma:

```
let user = {
  name: "John",
  age: 30,
}
```

That is called a “trailing” or “hanging” comma. Makes it easier to add/remove/move around properties, because all lines become alike.

Square brackets

For multiword properties, the dot access doesn’t work:

```
// this would give a syntax error
user.likes birds = true
```

JavaScript doesn’t understand that. It thinks that we address `user.likes`, and then gives a syntax error when comes across unexpected `birds`.

The dot requires the key to be a valid variable identifier. That implies: contains no spaces, doesn’t start with a digit and doesn’t include special characters (`$` and `_` are allowed).

There’s an alternative “square bracket notation” that works with any string:

```
let user = {};

// set
user["likes birds"] = true;

// get
alert(user["likes birds"]); // true

// delete
delete user["likes birds"];
```

Now everything is fine. Please note that the string inside the brackets is properly quoted (any type of quotes will do).

Square brackets also provide a way to obtain the property name as the result of any expression – as opposed to a literal string – like from a variable as follows:

```
let key = "likes birds";

// same as user["likes birds"] = true;
user[key] = true;
```

Here, the variable `key` may be calculated at run-time or depend on the user input. And then we use it to access the property. That gives us a great deal of flexibility.

For instance:

```
let user = {
  name: "John",
  age: 30
};

let key = prompt("What do you want to know about the user?", "name");

// access by variable
alert( user[key] ); // John (if enter "name")
```

The dot notation cannot be used in a similar way:

```
let user = {
  name: "John",
  age: 30
};

let key = "name";
alert( user.key ) // undefined
```

Computed properties

We can use square brackets in an object literal, when creating an object. That's called *computed properties*.

For instance:

```
let fruit = prompt("Which fruit to buy?", "apple");

let bag = {
  [fruit]: 5, // the name of the property is taken from the variable fru
};

alert( bag.apple ); // 5 if fruit="apple"
```

The meaning of a computed property is simple: `[fruit]` means that the property name should be taken from `fruit`.

So, if a visitor enters `"apple"`, `bag` will become `{apple: 5}`.

Essentially, that works the same as:

```
let fruit = prompt("Which fruit to buy?", "apple");
let bag = {};

// take property name from the fruit variable
bag[fruit] = 5;
```

…But looks nicer.

We can use more complex expressions inside square brackets:

```
let fruit = 'apple';
let bag = {
  [fruit + 'Computers']: 5 // bag.appleComputers = 5
};
```

Square brackets are much more powerful than dot notation. They allow any property names and variables. But they are also more cumbersome to write.

So most of the time, when property names are known and simple, the dot is used. And if we need something more complex, then we switch to square brackets.

Property value shorthand

In real code, we often use existing variables as values for property names.

For instance:

```
function makeUser(name, age) {
  return {
    name: name,
    age: age,
    // ...other properties
  };
}

let user = makeUser("John", 30);
alert(user.name); // John
```

In the example above, properties have the same names as variables. The use-case of making a property from a variable is so common, that there’s a special *property value shorthand* to make it shorter.

Instead of `name:name` we can just write `name` , like this:

```
function makeUser(name, age) {
  return {
    name, // same as name: name
    age,  // same as age: age
    // ...
  };
}
```

We can use both normal properties and shorthands in the same object:

```
let user = {
  name,  // same as name:name
  age: 30
};
```

Property names limitations

As we already know, a variable cannot have a name equal to one of the language-reserved words like “for”, “let”, “return” etc.

But for an object property, there's no such restriction:

```
// these properties are all right
let obj = {
  for: 1,
  let: 2,
  return: 3
};

alert( obj.for + obj.let + obj.return );  // 6
```

In short, there are no limitations on property names. They can be any strings or symbols (a special type for identifiers, to be covered later).

Other types are automatically converted to strings.

For instance, a number `0` becomes a string `"0"` when used as a property key:

```
let obj = {
  0: "test" // same as "0": "test"
};

// both alerts access the same property (the number 0 is converted to st
alert( obj["0"] ); // test
alert( obj[0] ); // test (same property)
```

There's a minor gotcha with a special property named `__proto__`. We can't set it to a non-object value:

```
let obj = {};
obj.__proto__ = 5; // assign a number
alert(obj.__proto__); // [object Object] - the value is an object, didn'
```

As we see from the code, the assignment to a primitive `5` is ignored.

We'll cover the special nature of `__proto__` in subsequent chapters, and suggest the ways to fix such behavior.

Property existence test, "in" operator

A notable feature of objects in JavaScript, compared to many other languages, is that it's possible to access any property. There will be no error if the property doesn't exist!

Reading a non-existing property just returns `undefined`. So we can easily test whether the property exists:

```
let user = {};

```

```
alert( user.noSuchProperty === undefined ); // true means "no such prope
```

There’s also a special operator `"in"` for that.

The syntax is:

```
"key" in object
```

For instance:

```
let user = { name: "John", age: 30 };

alert( "age" in user ); // true, user.age exists
alert( "blabla" in user ); // false, user.blabla doesn't exist
```

Please note that on the left side of `in` there must be a *property name*. That’s usually a quoted string.

If we omit quotes, that means a variable should contain the actual name to be tested. For instance:

```
let user = { age: 30 };

let key = "age";
alert( key in user ); // true, property "age" exists
```

Why does the `in` operator exist? Isn’t it enough to compare against `undefined`?

Well, most of the time the comparison with `undefined` works fine. But there’s a special case when it fails, but `"in"` works correctly.

It’s when an object property exists, but stores `undefined`:

```
let obj = {
  test: undefined
};

alert( obj.test ); // it's undefined, so - no such property?

alert( "test" in obj ); // true, the property does exist!
```

In the code above, the property `obj.test` technically exists. So the `in` operator works right.

Situations like this happen very rarely, because `undefined` should not be explicitly assigned. We mostly use `null` for “unknown” or “empty” values. So the `in` operator is an exotic guest in the code.

The "for..in" loop

To walk over all keys of an object, there exists a special form of the loop: `for..in`. This is a completely different thing from the `for(;;)` construct that we studied before.

The syntax:

```
for (key in object) {
  // executes the body for each key among object properties
}
```

For instance, let's output all properties of `user`:

```
let user = {
  name: "John",
  age: 30,
  isAdmin: true
};

for (let key in user) {
  // keys
  alert( key );  // name, age, isAdmin
  // values for the keys
  alert( user[key] ); // John, 30, true
}
```

Note that all "for" constructs allow us to declare the looping variable inside the loop, like `let key` here.

Also, we could use another variable name here instead of `key`. For instance, `"for (let prop in obj)"` is also widely used.

Ordered like an object

Are objects ordered? In other words, if we loop over an object, do we get all properties in the same order they were added? Can we rely on this?

The short answer is: "ordered in a special fashion": integer properties are sorted, others appear in creation order. The details follow.

As an example, let's consider an object with the phone codes:

```
let codes = {
  "49": "Germany",
  "41": "Switzerland",
  "44": "Great Britain",
  // ..,
  "1": "USA"
};
```

```

for (let code in codes) {
  alert(code); // 1, 41, 44, 49
}
```

The object may be used to suggest a list of options to the user. If we're making a site mainly for a German audience then we probably want `49` to be the first.

But if we run the code, we see a totally different picture:

- USA (1) goes first
- then Switzerland (41) and so on.

The phone codes go in the ascending sorted order, because they are integers. So we see `1, 41, 44, 49`.

Integer properties? What's that?

The "integer property" term here means a string that can be converted to-and-from an integer without a change.

So, `"49"` is an integer property name, because when it's transformed to an integer number and back, it's still the same. But `"+49"` and `"1.2"` are not:

```
// Number(...) explicitly converts to a number
// Math.trunc is a built-in function that removes the decimal part
alert( String(Math.trunc(Number("49"))) ); // "49", same, integer pr
alert( String(Math.trunc(Number("+49"))) ); // "49", not same "+49"
alert( String(Math.trunc(Number("1.2"))) ); // "1", not same "1.2"
```

…On the other hand, if the keys are non-integer, then they are listed in the creation order, for instance:

```
let user = {
  name: "John",
  surname: "Smith"
};
user.age = 25; // add one more

// non-integer properties are listed in the creation order
for (let prop in user) {
  alert( prop ); // name, surname, age
}
```

So, to fix the issue with the phone codes, we can "cheat" by making the codes non-integer. Adding a plus `"+"` sign before each code is enough.

Like this:

```
let codes = {
  "+49": "Germany",
  "+41": "Switzerland",
  "+44": "Great Britain",
  // ..,
  "+1": "USA"
};

for (let code in codes) {
  alert( +code ); // 49, 41, 44, 1
}
```

Now it works as intended.

Summary

Objects are associative arrays with several special features.

They store properties (key-value pairs), where:

- Property keys must be strings or symbols (usually strings).
- Values can be of any type.

To access a property, we can use:

- The dot notation: `obj.property` .
- Square brackets notation `obj["property"]` . Square brackets allow taking the key from a variable, like `obj[varWithKey]` .

Additional operators:

- To delete a property: `delete obj.prop` .
- To check if a property with the given key exists: `"key" in obj` .
- To iterate over an object: `for (let key in obj)` loop.

What we've studied in this chapter is called a "plain object", or just `Object` .

There are many other kinds of objects in JavaScript:

- `Array` to store ordered data collections,
- `Date` to store the information about the date and time,
- `Error` to store the information about an error.
- …And so on.

They have their special features that we'll study later. Sometimes people say something like "Array type" or "Date type", but formally they are not types of their own, but belong to a single "object" data type. And they extend it in various ways.

Objects in JavaScript are very powerful. Here we've just scratched the surface of a topic that is really huge. We'll be closely working with objects and learning more about them in further parts of the tutorial.

Tasks

Hello, object

importance: 5

Write the code, one line for each action:

1. Create an empty object `user`.
2. Add the property `name` with the value `John`.
3. Add the property `surname` with the value `Smith`.
4. Change the value of the `name` to `Pete`.
5. Remove the property `name` from the object.

solution

Check for emptiness

importance: 5

Write the function `isEmpty(obj)` which returns `true` if the object has no properties, `false` otherwise.

Should work like that:

```
let schedule = {};

alert( isEmpty(schedule) ); // true

schedule["8:30"] = "get up";

alert( isEmpty(schedule) ); // false
```

Open a sandbox with tests.

solution

Sum object properties

importance: 5

We have an object storing salaries of our team:

```
let salaries = {
  John: 100,
  Ann: 160,

  Pete: 130
}
```

Write the code to sum all salaries and store in the variable `sum`. Should be `390` in the example above.

If `salaries` is empty, then the result must be `0`.

solution

Multiply numeric property values by 2

importance: 3

Create a function `multiplyNumeric(obj)` that multiplies all numeric property values of `obj` by `2`.

For instance:

```
// before the call
let menu = {
  width: 200,
  height: 300,
  title: "My menu"
};

multiplyNumeric(menu);

// after the call
menu = {
  width: 400,
  height: 600,
  title: "My menu"
};
```

Please note that `multiplyNumeric` does not need to return anything. It should modify the object in-place.

P.S. Use `typeof` to check for a number here.

Open a sandbox with tests.

solution

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)

302 Comments 1 Login

G Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

♡ 111 Share

Best Newest Oldest

GiornoKujo
6 years ago

lots of info here, good tutorial!
Also, given the number of comments it looks as if people gave up already by now? :)

43 0 Reply

Farai Tanekha GiornoKujo
6 years ago

No way! I'm still going after 1am. I can't stop myself. This site is rich with info like you said.

43 0 Reply

RK **Rajwinder Kaur** Farai Tanekha
4 years ago

Absolutely right!

2 0 Reply

John Doe GiornoKujo
5 years ago

Hehe I came here for one thing. I want the array.push() equivalent in Objects. but oops I found a rabbit hole

3 0 Reply

GiornoKujo John Doe
5 years ago

.push for objects doesn't make sense because object properties are not ordered

5 3 Reply

danman1979 GiornoKujo
5 years ago

Of course they are. Did you read the tutorial?

'The short answer is: “ordered in a special fashion”: integer properties are sorted, others appear in creation order.'

11 0 Reply

jhorsager danman1979
15 days ago

Any push operation doesn't gaf about the order either way.

0 0 Reply

ridwan rais GiornoKujo
5 years ago edited

Object's properties are ordered depending on the creation order (except 'integer properties') though?
To add a new property, we can just use:

```
object.newProperty = value;
```

The new property should be added to the last order of the object. We can check with for...in

3 2 Reply

S

shashankh ridwan rais
2 years ago

Object.defineProperty(object_name,property,{value : "your_value"});

0 0 Reply

jhorsager GiornoKujo
15 days ago

When you're pushing something onto the stack it makes no difference what order the stack is in.

0 0 Reply

I

idk tho GiornoKujo
3 years ago

Is there not a way to add properties to objects that have already been declared? I assume there is, and I assume that works approximately the same way .push does. I could just wait and find out, but I could also generate dialogue ... in a 2-year-old thread. :/
Oh, well. Clicking the button.

0 1 Reply

T

Taughtly Anerd idk tho
3 years ago

It's shown above, you can add properties to the object whenever you feel like it.

View — uploads.disquscdn.com

1 0 Reply

M

Md Mahian Taughtly Anerd

a year ago

Right now

0 0 Reply

R

Rob GiornoKujo

5 years ago

It depends, this tutorial was exceptional up until the tests tutorial. I couldn't get any of the examples to work as they were not complete and much of the testing suite would not load in the browser. Now we are being asked to write tests that we couldn't learn. It could possibly be a deal breaker if the testing is not addressed properly. This is a beginner course, we don't want to be tweaking code to get libraries to load if we dont' know how to do that.

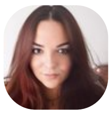
5 5 Reply

Joel Flores Rob

4 years ago

you need **return obj** in the function otherwise you wont see the results when you call the function

1 1 Reply



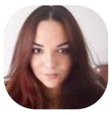
Eva Lavinia Bucur Joel Flores

4 years ago

> Please note that multiplyNumeric does not need to return anything. It should modify the object in-place.

6 0 Reply

This comment was deleted.



Eva Lavinia Bucur Guest

2 years ago

If the value of the key is a number, you should double it. If it is not a number, you should let it be as it is. Consider:

```
const arr = {
'one': 100,
'two': 200,
'three': 'three hundreds'
}
```

for (let prop in arr)
if (typeof arr[prop] == 'number')
arr[prop] *= 2

3 0 Reply

J

Julzedz Eva Lavinia Bucur
2 years ago

Thank you Eva. I now realize this is how numbers are recognized by JS I guess, as a string - 'number'. Data types like null, and undefined aren't required to be in strings to be recognized by JS. Hence, you could say xyz == null, not xyz == 'null'.

1 0 Reply

steveoncaffeine Julzedz
2 years ago

I don't think null can be used in that manner:

```
let item = null;
console.log(typeof(item) == null); // displays false
console.log(typeof(item) == 'null'); // Also displays false
```

Although undefined can be used that way, but it is required to be wrapped in quotes:

```
let item = undefined;
console.log(typeof(item) == undefined); // displays false
console.log(typeof(item) == "undefined"); // displays true
```

2 0 Reply

K

Kishore Baalaji steveoncaffeine
a year ago

Broo why are you adding parenthesis after typeof?? It is an operator not a function!!

1 1 Reply

B

brahim Kishore Baalaji
8 months ago

you know that parenthesis arent always for function par exmeple you can use them to for a phrase like this:
```typeof(our variable) ```
and its totally different than saying
``typeof our variable``

0 0 Reply

AN

Amir Nassaji steveoncaffeine
7 months ago

object not null.but null is not object type and it is a mistake existed in js from old to now

0 0 Reply

KN

karan nakra ➦ Eva Lavinia Bucur

a year ago

I think your solution will fail in this case
const arr = {
'one': 100,
'two': 200,
'three': 'three hundreds',
"four":NaN;
}
since typeof NaN = "number"
arr[prop] * 2 = NaN;;

0 0 Reply

Eva Lavinia Bucur ➦ karan nakra

a year ago

It actually won't, 'cause the problem stated that you have to double the key's value if the original value is a number and you have to leave the value untouched, if the original value is not a number. So { 'four': NaN }, indeed, would get processed for the value 'four'. But the original value would be NaN, which, even if being a number value, doubled would still be NaN. So, in the end, the resulting value would be equal to the original value. Take it as you'd like: NaN is still the original value left untouched and, at the same time, is the double of the original NaN value.

2 0 Reply

Kattens ➦ Guest

2 years ago

when you use the typeof fxn it returns the data type as a 'number ', 'string' etc when displayed with an alert. so we compare with the "==" if the value of the key is same as 'number'. meaning both will look like 'number' == ' number' for the statement to be true

0 0 Reply

M

Maryanne ➦ Eva Lavinia Bucur

2 years ago

I returned mine

```
function multiplyNumeric(menu){
for (const prop in menu){
if (typeof menu[prop] === "number") menu[prop] *= 2;
}
```

return menu;
}

console.log(multiplyNumeric(menu))

0 1 Reply

J

Jerry Nwosu Joel Flores

3 years ago

You could access the object's properties after calling the function.. like so;

```
let menu = {
width: 200,
height: 300,
title: "My menu",
};

function multiplyNumeric(obj) {
for (let key in obj) {
if (typeof obj[key] === "number") {
obj[key] *= 2;
}
}
}

multiplyNumeric(menu);
console.log(menu.width);
```

2 0 Reply

Joel Flores Joel Flores

4 years ago

function multiplyNumeric(obj) {
for(let key in obj) {
if(typeof(obj[key]) == 'number'){
obj[key] *= 2
}
}
return obj
}

0 0 Reply

A

Artur Geerkhanov Joel Flores

4 years ago

It's a side effect function. No need to return anything here.

1 1 Reply

S

sururat itunuoluwa lawal Artur Geerkhanov

3 years ago

I still don't get why it shouldn't

0 0 Reply

SaEeD sururat itunuoluwa lawal

2 years ago

the code in the function(after calling the function) would change the object in place so it wont need return the new object.
remember the topic about global variable and local variable in function section. when you did change the global variable in a function you dont need to return anything. you need to return the object in function when you created object in that function. but in this excercise the object is global and after calling the function it would change that object.

1 0 Reply

U

Unknown io Rob

2 years ago

To be fair you can learn testing anytime.

0 0 Reply

the 1stgeek GiornoKujo

4 years ago

i'm still going

0 0 Reply

Adarsh Naidu GiornoKujo

5 years ago

The previous comments do not appear for some reason.

0 0 Reply

🍓Calvin dirty slut🔞

2 years ago

Hehe Looks good

15 0 Reply

Eva Lavinia Bucur

4 years ago edited

I'd like to add that all objects passed as arguments to functions are passed *by reference* (meaning "whatever changes happens inside the function will be reflected in the outer scope for the object passed as an argument").

Example:

```
let user = {
    name: 'Newbie',
    age: 21
}

function changeUserName(userObject) {
    userObject.name = 'Advanced';
}

changeUserName(user);

console.log(user.name);
```

see more

7 0 Reply

M

Matt Thompson Eva Lavinia Bucur

14 days ago

Thank you, I spent some time figuring out why it appeared sometimes arguments were passed by reference vs value and it did not occur to me objects are by reference and primitives are by value until I I read your reply.

0 0 Reply

S

sururat itunuoluwa lawal Eva Lavinia Bucur

3 years ago

It's very detailed, I'm loving this

0 0 Reply

BFonseca Eva Lavinia Bucur

3 years ago

Chill out the next chapter make that clear

0 0 Reply

Bobby Chicano

3 years ago

Some of these tests are just plain bad. I understand we need to struggle to learn, but the gap between what is being taught and the mental gymnastics you need to accomplish to get to some of these answers is too much. It ends up upsetting me, because after a long struggle I have to reveal the solution only to realize I never would have thought of it that way. Maybe if I was an experienced coder, but definitely not someone just learning the language. Smh.

9 1 Reply

S

someuniqueperson Bobby Chicano

2 years ago

I agree.

1 0 Reply

VERCRAZY

VERCRAZY

5 years ago edited

Arrow function syntax for the last question:

```
let multiplyNumeric = (obj) => {
  for (let key in obj) {
    if (!isNaN(obj[key])) {
      obj[key] *= 2
    };
  }
}
```

I used "!isNaN()" rather than "typeof obj[key] == 'number'" because it allows "number-like" values stored as strings to still be multiplied by 2.

For example.

menu.width = "100"

Would still be multiplied by 2 with the "!isNaN()" evaluation, whereas it would not with the "typeof()" evaluation.

11 2 Reply

Vic VERCRAZY

3 years ago edited

Bad decision about isNaN, because if there will be Boolean value in an object, then that one will be affected as well:

```
isNaN(true) // false
true * 2 // 2
isNaN(false) // false
false*2 // 0

menu = {
width: 400,
height: 600,
title: "My menu"
isShowed: 2; // was true, but was affected by isNaN implementation
};
```

I guess better is to keep it with "typeof"

2 0 Reply

M

Mahmut ERDEM VERCRAZY

4 years ago

this is way too complicated. it scares me lol

1 0 Reply

S

SnakeTwix ↪ VERCRAZY

5 years ago

> I used "!isNaN()" rather than "typeof obj[key] == 'number'" because it allows "number-like" values stored as strings to still be multiplied by 2.

Well, what you did is good enough, but there is another built in function isFinite(). It returns when it's not an unfinite number and returns false when it's infinite or any other type.

1 1 Reply

T

Toby ↪ VERCRAZY

5 years ago

Are you sure of this? I tried it and it worked. Here's a link to the repl https://repl.it/repls/Chill...

0 0 Reply

C

Comrade 36_rus ↪ VERCRAZY

5 years ago

> Please note that multiplyNumeric does not need to return anything