

 → [The JavaScript language](#) → [Objects: the basics](#)

 31 May 2019

Object methods, "this"

Objects are usually created to represent entities of the real world, like users, orders and so on:

```
1 let user = {  
2   name: "John",  
3   age: 30  
4 };
```

And, in the real world, a user can *act*: select something from the shopping cart, login, logout etc.

Actions are represented in JavaScript by functions in properties.

Method examples

For the start, let's teach the `user` to say hello:

```
1 let user = {  
2   name: "John",  
3   age: 30  
4 };  
5  
6 user.sayHi = function() {  
7   alert("Hello!");  
8 };  
9  
10 user.sayHi(); // Hello!
```

Here we've just used a Function Expression to create the function and assign it to the property `user.sayHi` of the object.

Then we can call it. The user can now speak!

A function that is the property of an object is called its *method*.

So, here we've got a method `sayHi` of the object `user`.

Of course, we could use a pre-declared function as a method, like this:

```
1 let user = {  
2   // ...  
3 };  
4  
5 // first, declare  
6 function sayHi() {  
7   alert("Hello!");  
8 };  
9  
10 // then add as a method  
11 user.sayHi = sayHi;  
12  
13 user.sayHi(); // Hello!
```

Object-oriented programming

When we write our code using objects to represent entities, that's called an [object-oriented programming](#), in short: "OOP".

OOP is a big thing, an interesting science of its own. How to choose the right entities? How to organize the interaction between them? That's architecture, and there are great books on that topic, like "Design Patterns: Elements of Reusable Object-Oriented Software" by E.Gamma, R.Helm, R.Johnson, J.Vissides or "Object-Oriented Analysis and Design with Applications" by G.Booch, and more.

Method shorthand

There exists a shorter syntax for methods in an object literal:

```
1 // these objects do the same  
2  
3 user = {
```

```

4   sayHi: function() {
5       alert("Hello");
6   }
7 };
8
9 // method shorthand looks better, right?
10 user = {
11     sayHi() { // same as "sayHi: function()"
12         alert("Hello");
13     }
14 };

```

As demonstrated, we can omit `"function"` and just write `sayHi()`.

To tell the truth, the notations are not fully identical. There are subtle differences related to object inheritance (to be covered later), but for now they do not matter. In almost all cases the shorter syntax is preferred.

“this” in methods

It's common that an object method needs to access the information stored in the object to do its job.

For instance, the code inside `user.sayHi()` may need the name of the `user`.

To access the object, a method can use the `this` keyword.

The value of `this` is the object “before dot”, the one used to call the method.

For instance:

```

1  let user = {
2      name: "John",
3      age: 30,
4
5      sayHi() {
6          alert(this.name);
7      }
8
9  };
10
11 user.sayHi(); // John

```

Here during the execution of `user.sayHi()`, the value of `this` will be `user`.

Technically, it's also possible to access the object without `this`, by referencing it via the outer variable:

```

1  let user = {
2      name: "John",
3      age: 30,
4
5      sayHi() {
6          alert(user.name); // "user" instead of "this"
7      }
8
9  };

```

...But such code is unreliable. If we decide to copy `user` to another variable, e.g. `admin = user` and overwrite `user` with something else, then it will access the wrong object.

That's demonstrated below:

```

1  let user = {
2      name: "John",
3      age: 30,
4
5      sayHi() {
6          alert( user.name ); // leads to an error
7      }
8
9  };
10
11
12 let admin = user;
13 user = null; // overwrite to make things obvious
14
15 admin.sayHi(); // Whoops! inside sayHi(), the old name is used! error!

```

If we used `this.name` instead of `user.name` inside the `alert`, then the code would work.

“this” is not bound

In JavaScript, “this” keyword behaves unlike most other programming languages. It can be used in any function.

There's no syntax error in the code like that:

```
1 function sayHi() {
2   alert( this.name );
3 }
```

The value of `this` is evaluated during the run-time, depending on the context. And it can be anything.

For instance, here the same function is assigned to two different objects and has different "this" in the calls:

```
1 let user = { name: "John" };
2 let admin = { name: "Admin" };
3
4 function sayHi() {
5   alert( this.name );
6 }
7
8 // use the same function in two objects
9 user.f = sayHi;
10 admin.f = sayHi;
11
12 // these calls have different this
13 // "this" inside the function is the object "before the dot"
14 user.f(); // John (this == user)
15 admin.f(); // Admin (this == admin)
16
17 admin['f'](); // Admin (dot or square brackets access the method – doesn't matter)
```

The rule is simple: if `obj.f()` is called, then `this` is `obj` during the call of `f`. So it's either `user` or `admin` in the example above.

i Calling without an object: `this == undefined`

We can even call the function without an object at all:

```
1 function sayHi() {
2   alert(this);
3 }
4
5 sayHi(); // undefined
```

In this case `this` is `undefined` in strict mode. If we try to access `this.name`, there will be an error.

In non-strict mode the value of `this` in such case will be the *global object* (`window` in a browser, we'll get to it later in the chapter [Global object](#)). This is a historical behavior that "use strict" fixes.

Usually such call is an programming error. If there's `this` inside a function, it expects to be called in an object context.

i The consequences of unbound `this`

If you come from another programming language, then you are probably used to the idea of a "bound `this`", where methods defined in an object always have `this` referencing that object.

In JavaScript `this` is "free", its value is evaluated at call-time and does not depend on where the method was declared, but rather on what's the object "before the dot".

The concept of run-time evaluated `this` has both pluses and minuses. On the one hand, a function can be reused for different objects. On the other hand, greater flexibility opens a place for mistakes.

Here our position is not to judge whether this language design decision is good or bad. We'll understand how to work with it, how to get benefits and evade problems.

Internals: Reference Type

⚠ In-depth language feature

This section covers an advanced topic, to understand certain edge-cases better.

If you want to go on faster, it can be skipped or postponed.

An intricate method call can lose `this`, for instance:

```
1 let user = {
2   name: "John",
3   hi() { alert(this.name); },
4   bye() { alert("Bye"); }
5 };
```

```

6
7 user.hi(); // John (the simple call works)
8
9 // now let's call user.hi or user.bye depending on the name
10 (user.name == "John" ? user.hi : user.bye()); // Error!

```

On the last line there is a ternary operator that chooses either `user.hi` or `user.bye`. In this case the result is `user.hi`.

The method is immediately called with parentheses `()`. But it doesn't work right!

You can see that the call results in an error, because the value of `"this"` inside the call becomes `undefined`.

This works (object dot method):

```
1 user.hi();
```

This doesn't (evaluated method):

```
1 (user.name == "John" ? user.hi : user.bye()); // Error!
```

Why? If we want to understand why it happens, let's get under the hood of how `obj.method()` call works.

Looking closely, we may notice two operations in `obj.method()` statement:

1. First, the dot `'.'` retrieves the property `obj.method`.
2. Then parentheses `()` execute it.

So, how does the information about `this` get passed from the first part to the second one?

If we put these operations on separate lines, then `this` will be lost for sure:

```

1 let user = {
2   name: "John",
3   hi() { alert(this.name); }
4 }
5
6 // split getting and calling the method in two lines
7 let hi = user.hi;
8 hi(); // Error, because this is undefined

```

Here `hi = user.hi` puts the function into the variable, and then on the last line it is completely standalone, and so there's no `this`.

To make `user.hi()` calls work, JavaScript uses a trick – the dot `'.'` returns not a function, but a value of the special [Reference 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.
- `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:

```

1 // Reference Type value
2 (user, "hi", true)

```

When parentheses `()` are called on the Reference Type, they receive the full information about the object and its method, and can set the right `this` (`=user` in this case).

Any other operation like assignment `hi = user.hi` discards the reference type as a whole, takes the value of `user.hi` (a function) and passes it on. So any further operation "loses" `this`.

So, as the result, the value of `this` is only passed the right way if the function is called directly using a dot `obj.method()` or square brackets `obj['method']()` syntax (they do the same here). Later in this tutorial, we will learn various ways to solve this problem such as [func.bind\(\)](#).

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:

```

1 let user = {
2   firstName: "Ilya",
3   sayHi() {
4     let arrow = () => alert(this.firstName);
5     arrow();
6   }
7 };
8
9 user.sayHi(); // Ilya

```

That's a special feature of arrow functions, it's useful when we actually do not want to have a separate `this`, but rather to take it from the outer context. Later in the chapter [Arrow functions revisited](#) we'll go more deeply into arrow functions.

Summary

- Functions that are stored in object properties are called "methods".
- Methods allow objects to "act" like `object.doSomething()`.
- Methods can reference the object as `this`.

The value of `this` is defined at run-time.

- When a function is declared, it may use `this`, but that `this` has no value until the function is called.
- That function can be copied between objects.
- When a function is called in the "method" syntax: `object.method()`, the value of `this` during the call is `object`.

Please note that arrow functions are special: they have no `this`. When `this` is accessed inside an arrow function, it is taken from outside.

✓ Tasks

Syntax check

importance: 2

What is the result of this code?

```

1 let user = {
2   name: "John",
3   go: function() { alert(this.name) }
4 }
5
6 (user.go)()

```

P.S. There's a pitfall :)

[solution](#)

Explain the value of "this"

importance: 3

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?

```

1 let obj, method;
2
3 obj = {
4   go: function() { alert(this); }
5 };
6
7 obj.go();           // (1) [object Object]
8
9 (obj.go)();         // (2) [object Object]
10
11 (method = obj.go)(); // (3) undefined
12
13 (obj.go || obj.stop)(); // (4) undefined

```

[solution](#)

Using "this" in object literal [↗](#)

importance: 5

Here the function `makeUser` returns an object.

What is the result of accessing its `ref` ? Why?

```
1 function makeUser() {
2   return {
3     name: "John",
4     ref: this
5   };
6 };
7
8 let user = makeUser();
9
10 alert( user.ref.name ); // What's the result?
```

[solution](#)

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.

```
1 let calculator = {
2   // ... your code ...
3 };
4
5 calculator.read();
6 alert( calculator.sum() );
7 alert( calculator.mul() );
```

[Run the demo](#)

[Open a sandbox with tests.](#)

[solution](#)

Chaining [↗](#)

importance: 2

There's a `ladder` object that allows to go up and down:

```
1 let ladder = {
2   step: 0,
3   up() {
4     this.step++;
5   },
6   down() {
7     this.step--;
8   },
9   showStep: function() { // shows the current step
10    alert( this.step );
11  }
12 };
```

Now, if we need to make several calls in sequence, can do it like this:

```
1 ladder.up();
2 ladder.up();
3 ladder.down();
4 ladder.showStep(); // 1
```

Modify the code of `up`, `down` and `showStep` to make the calls chainable, like this:

```
1 ladder.up().up().down().showStep(); // 1
```

Such approach is widely used across JavaScript libraries.

[Open a sandbox with tests.](#)

solution



Previous lesson

Next lesson



Share  

 [Tutorial map](#)

Comments

- You're welcome to post additions, questions to the articles and answers to them.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox ([plnkr](#), [JSBin](#), [codepen](#)...)
- If you can't understand something in the article – please elaborate.