Object methods, "this"

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

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

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:

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

user.sayHi = function() {
  alert("Hello!");
};

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:

```
let user = {
  // ...
};

// first, declare
function sayHi() {
  alert("Hello!");
};

// then add as a method
user.sayHi = sayHi;

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. We'll scratch the surface of that topic later in the chapter Objects, classes, inheritance.

Method shorthand

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

```
// these objects do the same

let user = {
  sayHi: function() {
    alert("Hello");
  }
};

// method shorthand looks better, right?
let user = {
  sayHi() { // same as "sayHi: function()"
    alert("Hello");
  }
};
```

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:

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

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

};

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:

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

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

};
```

...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:

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

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

};


let admin = user;
user = null; // overwrite to make things obvious

```

```
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. First, it can be used in any function.

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

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

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

For instance, the same function may have different "this" when called from different objects:

```
let user = { name: "John" };
let admin = { name: "Admin" };

function sayHi() {
  alert( this.name );
}

// use the same functions in two objects
user.f = sayHi;
admin.f = sayHi;

// these calls have different this
// "this" inside the function is the object "before the dot"
user.f(); // John  (this == user)
admin.f(); // Admin  (this == admin)

admin['f'](); // Admin (dot or square brackets access the method – doesn't matter)
```

Actually, we can call the function without an object at all:

```
function sayHi() {
  alert(this);
}

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 (if one forgets `use strict`) the value of `this` in such case will be the *global object* (`window` in a browser, we'll get to it later). This is a historical behavior that `"use strict"` fixes.

Please note that usually a call of a function that uses `this` without an object is not normal, but rather a programming mistake. If a function has `this`, then it is usually meant to be called in the context of an object.

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:

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

user.hi(); // John (the simple call works)

// now let's call user.hi or user.bye depending on the name
(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):

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

This doesn't (evaluated method):

```
(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:

```
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
```

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:

```
// Reference Type value
(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:

```
let user = {
  firstName: "Ilya",
  sayHi() {
    let arrow = () => alert(this.firstName);
    arrow();
  }
};

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?

```
let user = {
  name: "John",
  go: function() { alert(this.name) }
}

(user.go)()
```

P.S. There's a pitfall :)

solution

Error!

Try it:

```
let user = {
  name: "John",
  go: function() { alert(this.name) }
}

(user.go)() // error!
```

The error message in most browsers does not give understanding what went wrong.

The error appears because a semicolon is missing after `user = {...}`.

JavaScript does not assume a semicolon before a bracket `(user.go)()`, so it reads the code like:

```
let user = { go:... }(user.go)()
```

Then we can also see that such a joint expression is syntactically a call of the object `{ go: ... }` as a function with the argument `(user.go)`. And that also happens on the same line with `let user`, so the `user` object has not yet even been defined, hence the error.

If we insert the semicolon, all is fine:

```
let user = {
  name: "John",
  go: function() { alert(this.name) }
};

(user.go)() // John
```

Please note that brackets around `(user.go)` do nothing here. Usually they setup the order of operations, but here the dot `.` works first anyway, so there's no effect. Only the semicolon thing matters.

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?

```
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
```

solution

Here's the explanations.

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:

```
f = obj.go; // calculate the expression
f();        // call what we have
```

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

4. 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`.

Using "this" in object literal

importance: 5

Here the function `makeUser` returns an object.

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

```
function makeUser() {
  return {
    name: "John",
    ref: this
  };
};

let user = makeUser();

alert( user.ref.name ); // What's the result?
```

solution

Answer: an error.

Try it:

```
function makeUser() {
  return {
    name: "John",
    ref: this
  };
};

let user = makeUser();

alert( user.ref.name ); // Error: Cannot read property 'name' of undefined
```

That's because rules that set `this` do not look at object literals.

Here the value of `this` inside `makeUser()` is `undefined`, because it is called as a function, not as a method.

And the object literal itself has no effect on `this`. The value of `this` is one for the whole function, code blocks and object literals do not affect it.

So `ref: this` actually takes current `this` of the function.

Here's the opposite case:

```
function makeUser() {
  return {
    name: "John",
    ref() {
      return this;
    }
  };
};
```

```
let user = makeUser();

alert( user.ref().name ); // John
```

Now it works, because `user.ref()` is a method. And the value of `this` is set to the object before dot `.`.

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() );
```

Run the demo

Open a sandbox with tests.

solution

```
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() );
```

Open the solution with tests in a sandbox.

Chaining

importance: 2

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

```
let ladder = {
  step: 0,
  up() {
    this.step++;
  },
  down() {
    this.step--;
  },
  showStep: function() { // shows the current step
    alert( this.step );
```

```
  }
};
```

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

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

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

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

Such approach is widely used across JavaScript libraries.

Open a sandbox with tests.

solution

The solution is to return the object itself from every call.

```
let ladder = {
  step: 0,
  up() {
    this.step++;
    return this;
  },
  down() {
    this.step--;
    return this;
  },
  showStep() {
    alert( this.step );
    return this;
  }
}

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

We also can write a single call per line. For long chains it's more readable:

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

Open the solution with tests in a sandbox.

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...)