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 a 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 a function and assign it to the property `user.sayHi` of the object.

Then we can call it as `user.sayHi()`. The user can now speak!

A function that is a 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 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:

```
// these objects do the same

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

// method shorthand looks better, right?
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() {
    // "this" is the "current object"
    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

admin.sayHi(); // TypeError: Cannot read property 'name' of null
```

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

“this” is not bound

In JavaScript, keyword `this` behaves unlike most other programming languages. It can be used in any function, even if it’s not a method of an object.

There’s no syntax error in the following example:

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

The value of `this` is evaluated during the run-time, depending on the context.

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

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

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

// use the same function 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 – does
```

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.

Calling without an object: `this == undefined`

We can even 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 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 a programming error. If there’s `this` inside a function, it expects to be called in an object context.

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 object is “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, the greater flexibility creates more possibilities 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 avoid problems.

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.
- A 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.

Share

Edit on GitHub

Tasks

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

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

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

Comments

read this before commenting…