Javascript Day - 7

Object methods, “this”, Constructor Function, Optional Chaining

Object methods

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

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.

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

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

user.sayHi(); // Hello!
```

Object methods

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

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

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

user.sayHi(); // Hello!
```

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.

Object methods(method shorthand)

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

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

In almost all cases, the shorter syntax is preferred.

```
// these objects do the same

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

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

Object methods(this keyword)

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.

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

Object methods(this keyword)

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

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.

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

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

};
```

Object methods(this keyword)

If we used this.name instead of user.name inside the alert, then the code would 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(); // TypeError: Cannot read property 'name' of null
```

“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.

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

There’s no syntax error in the following example: =>

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:

```
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 - doesn't matter)
```

“This” is not bound

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

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

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

In **non-strict mode** the **value of this** in such case **will be the global object (window in a browser)**

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.

Arrow Function 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 arrow = () => {
  console.log(this);
}

let user = {
  name: "John",
  sayHi: arrow,
}

user.sayHi();
```

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

user.sayHi(); // Ilya
```

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.

Constructor function and “new” operator

The regular {...} syntax allows us to create one object. But often we need to create many similar objects, like multiple users or menu items and so on.

That can be done using constructor functions and the "new" operator.

Constructor functions technically are regular functions. There are two conventions though:

- They are named with capital letter first.
- They should be executed only with "new" operator.

```
1  function User(name) {
2    this.name = name;
3    this.isAdmin = false;
4  }
5
6  let user = new User("Jack");
7
8  alert(user.name); // Jack
9  alert(user.isAdmin); // false
```

Constructor function and “new” operator

When a function is executed with “new”, it does the following steps:

A new empty object is created and assigned to this.

The function body executes. Usually it modifies this, adds new properties to it.

The value of this is returned.

In other words, new User(...) does something like:

```
function User(name) {
  // this = {};  (implicitly)

  // add properties to this
  this.name = name;
  this.isAdmin = false;

  // return this;  (implicitly)
}
```

Constructor function and “new” operator

Let’s note once again – technically, any function (except arrow functions, as they don’t have this) can be used as a constructor. It can be run with new, and it will execute the algorithm above. The “capital letter first” is a common agreement, to make it clear that a function is to be run with new.

```
function User(name) {
  // this = {};  (implicitly)

  // add properties to this
  this.name = name;
  this.isAdmin = false;

  // return this;  (implicitly)
}
```

Return from constructor

Usually, constructors do not have a return statement. Their task is to write all necessary stuff into this, and it automatically becomes the result.

But if there is a return statement, then the rule is simple:

- If return is called with an object, then the object is returned instead of this.
- If return is called with a primitive, it's ignored. "this" is returned

Usually constructors don't have a return statement. Here we mention the special behavior with returning objects mainly for the sake of completeness.

```
function BigUser() {

  this.name = "John";

  return { name: "Godzilla" };  // <-- returns this object
}

alert( new BigUser().name );  // Godzilla, got that object
```

```
function SmallUser() {

  this.name = "John";

  return; // <-- returns this
}

alert( new SmallUser().name );  // John
```

Methods in constructor

Using constructor functions to create objects gives a great deal of flexibility. The constructor function may have parameters that define how to construct the object, and what to put in it.

Of course, we can add to this not only properties, but methods as well.

For instance, new User(name) below creates an object with the given name and the method sayHi:

```
function User(name) {
  this.name = name;

  this.sayHi = function() {
    alert( "My name is: " + this.name );
  };
}

let john = new User("John");

john.sayHi(); // My name is: John

/*
john = {
   name: "John",
   sayHi: function() { ... }
}
*/
```

Summary

- Constructor functions or, briefly, constructors, are regular functions, but there's a common agreement to name them with capital letter first.
- Constructor functions should only be called using **new**. Such a call implies a creation of empty **this** at the start and returning the populated one at the end.

We can use constructor functions to make multiple similar objects.

Optional Chaining `?.'

The optional chaining ?. is a safe way to access nested object properties, even if an intermediate property doesn't exist.

As an example, let's say we have user objects that hold the information about our users.

Most of our users have addresses in user.address property, with the street user.address.street, but some did not provide them.

In such case, when we attempt to get user.address.street, and the user happens to be without an address, we get an error:

```
1  let user = {}; // a user without "address" property
2
3  alert(user.address.street); // Error!
```

Optional Chaining `?.'

That's the expected result. JavaScript works like this. As user.address is undefined, an attempt to get user.address.street fails with an error.

In many practical cases we'd prefer to get undefined instead of an error here (meaning "no street").

How can we do this?

The obvious solution would be to check the value using if or the conditional operator ?, before accessing its property, like this:

```
1  let user = {};
2
3  alert(user.address ? user.address.street : undefined);
```

It works, there's no error... But it's quite inelegant. As you can see, the "user.address" appears twice in the code.

Optional Chaining `?.'

For more deeply nested properties, it becomes even uglier, as more repetitions are required.

E.g. let's get user.address.street.name in a similar fashion.

```
let user = {}; // user has no address

alert(user.address ? user.address.street ? user.address.street.name : null : null);
```

That's just awful, one may even have problems understanding such code.

There's a little better way to write it, using the && operator:

```
let user = {}; // user has no address

alert( user.address && user.address.street && user.address.street.name );
```

AND'ing the whole path to the property ensures that all components exist (if not, the evaluation stops), but also isn't ideal.

Optional Chaining '?.'

As you can see, property names are still duplicated in the code. E.g. in the code above, user.address appears three times.

That's why the optional chaining ?. was added to the language. To solve this problem once and for all!

The optional chaining ?. stops the evaluation if the value before ?. is undefined or null and returns undefined.

Here's the safe way to access user.address.street using ?.:

```
1  let user = {}; // user has no address
2
3  alert( user?.address?.street ); // undefined (no error)
```

Don’t overuse Optional Chaining `?.’

We should use ?. only where it’s ok that something doesn’t exist.

For example, if according to our code logic user object must exist, but address is optional, then we should write user.address?.street, but not user?.address?.street.

Then, if user happens to be undefined, we’ll see a programming error about it and fix it. Otherwise, if we overuse ?., coding errors can be silenced where not appropriate, and become more difficult to debug.

The variable before ?. must be declared

If there’s no variable user at all, then user?.anything triggers an error:

```
1  // ReferenceError: user is not defined
2  user?.address;
```

The variable must be declared (e.g. let/const/var user or as a function parameter). The optional chaining works only for declared variables.

Short Circuiting

As it was said before, the ?. immediately stops (“short-circuits”) the evaluation if the left part doesn’t exist.

So, if there are any further function calls or operations to the right of ?., they won’t be made.

```
let user = null;
let x = 0;

user?.sayHi(x++); // no "user", so the execution doesn't reach sayHi call and x++

alert(x); // 0, value not incremented
```

Other variants: ?.() or ?.[]

The optional chaining ?. is not an operator, but a special syntax construct, that also works with functions and square brackets.

For example, ?.() is used to call a function that may not exist.

In the code below, some of our users have admin method, and some don't:

```
let userAdmin = {
  admin() {
    alert("I am admin");
  }
};

let userGuest = {};

userAdmin.admin?.(); // I am admin

userGuest.admin?.(); // nothing happens (no such method)
```

Other variants: ?.() or ?.[]

The ?.[] syntax also works, if we'd like to use brackets [] to access properties instead of dot .. Similar to previous cases, it allows to safely read a property from an object that may not exist.

```
let key = "firstName";

let user1 = {
  firstName: "John"
};

let user2 = null;

alert( user1?.[key] ); // John
alert( user2?.[key] ); // undefined
```

Also we can use ?. with delete:

```
delete user?.name; // delete user.name if user exists
```

We can use ?. for safe reading and deleting, but not writing

The optional chaining ?. has no use on the left side of an assignment.

```
let user = null;

user?.name = "John"; // Error, doesn't work
// because it evaluates to: undefined = "John"
```

Summary

The optional chaining ?. syntax has three forms:

- obj?.prop – returns obj.prop if obj exists, otherwise undefined.
- obj?.[prop] – returns obj[prop] if obj exists, otherwise undefined.
- obj.method?.() – calls obj.method() if obj.method exists, otherwise returns undefined.

As we can see, all of them are straightforward and simple to use. The ?. checks the left part for null/undefined and allows the evaluation to proceed if it's not so.

A chain of ?. allows to safely access nested properties.

Still, we should apply ?. carefully, only where it's acceptable, according to our code logic, that the left part doesn't exist. So that it won't hide programming errors from us, if they occur.

Object Methods

For plain objects, the following methods are available:

- Object.keys(obj) – returns an array of keys.
- Object.values(obj) – returns an array of values.
- Object.entries(obj) – returns an array of [key, value] pairs.

The first difference is that we have to call Object.keys(obj), and not obj.keys().

Why so? The main reason is flexibility. Remember, objects are a base of all complex structures in JavaScript. So we may have an object of our own like data that implements its own data.values() method. And we still can call Object.values(data) on it.

Object Methods

- Object.keys(user) = ["name", "age"]
- Object.values(user) = ["John", 30]
- Object.entries(user) = [["name","John"], ["age",30]]

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

Here's an example of using Object.values to loop over property values:

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

// loop over values
for (let value of Object.values(user)) {
  alert(value); // John, then 30
}
```

Transforming Objects

Objects lack many methods that exist for arrays, e.g. map, filter and others.

If we'd like to apply them, then we can use Object.entries followed by Object.fromEntries

1. Use Object.entries(obj) to get an array of key/value pairs from obj.
2. Use array methods on that array, e.g. map, to transform these key/value pairs.
3. Use Object.fromEntries(array) on the resulting array to turn it back into an object.

```
let prices = {
  banana: 1,
  orange: 2,
  meat: 4,
};

let doublePrices = Object.fromEntries(
  // convert prices to array, map each key/value pair into another pair
  // and then fromEntries gives back the object
  Object.entries(prices).map(entry => [entry[0], entry[1] * 2])
);

alert(doublePrices.meat); // 8
```