17th September 2019

Constructor, operator "new"

The regular `{...}` syntax allows 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 function

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

1. They are named with capital letter first.
2. They should be executed only with `"new"` operator.

For instance:

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}

let user = new User("Jack");

alert(user.name); // Jack
alert(user.isAdmin); // false
```

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

1. A new empty object is created and assigned to `this`.
2. The function body executes. Usually it modifies `this`, adds new properties to it.
3. 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)
}
```

So `let user = new User("Jack")` gives the same result as:

```
let user = {
  name: "Jack",
  isAdmin: false
};
```

Now if we want to create other users, we can call `new User("Ann")`, `new User("Alice")` and so on. Much shorter than using literals every time, and also easy to read.

That's the main purpose of constructors – to implement reusable object creation code.

Let's note once again – technically, any function can be used as a constructor. That is: any function 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`.

new function() { ... }

If we have many lines of code all about creation of a single complex object, we can wrap them in constructor function, like this:

```
let user = new function() {
  this.name = "John";
  this.isAdmin = false;

  // ...other code for user creation
  // maybe complex logic and statements
  // local variables etc
};
```

The constructor can't be called again, because it is not saved anywhere, just created and called. So this trick aims to encapsulate the code that constructs the single object, without future reuse.

Constructor mode test: new.target

Advanced stuff

The syntax from this section is rarely used, skip it unless you want to know everything.

Inside a function, we can check whether it was called with `new` or without it, using a special `new.target` property.

It is empty for regular calls and equals the function if called with `new`:

```
function User() {
  alert(new.target);
}

```

```
// without "new":
User(); // undefined

// with "new":
new User(); // function User { ... }
```

That can be used inside the function to know whether it was called with `new`, "in constructor mode", or without it, "in regular mode".

We can also make both `new` and regular calls to do the same, like this:

```
function User(name) {
  if (!new.target) { // if you run me without new
    return new User(name); // ...I will add new for you
  }

  this.name = name;
}

let john = User("John"); // redirects call to new User
alert(john.name); // John
```

This approach is sometimes used in libraries to make the syntax more flexible. So that people may call the function with or without `new`, and it still works.

Probably not a good thing to use everywhere though, because omitting `new` makes it a bit less obvious what's going on. With `new` we all know that the new object is being created.

Return from constructors

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.

In other words, `return` with an object returns that object, in all other cases `this` is returned.

For instance, here `return` overrides `this` by returning an object:

```
function BigUser() {

  this.name = "John";

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

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

And here's an example with an empty `return` (or we could place a primitive after it, doesn't matter):

```
function SmallUser() {

  this.name = "John";

  return; // <-- returns this
}

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

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

Omitting parentheses

By the way, we can omit parentheses after `new`, if it has no arguments:

```
let user = new User; // <-- no parentheses
// same as
let user = new User();
```

Omitting parentheses here is not considered a "good style", but the syntax is permitted by specification.

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() { ... }
}
*/
```

To create complex objects, there's a more advanced syntax, classes, that we'll cover later.

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.

JavaScript provides constructor functions for many built-in language objects: like `Date` for dates, `Set` for sets and others that we plan to study.

> **Objects, we'll be back!**
>
> In this chapter we only cover the basics about objects and constructors. They are essential for learning more about data types and functions in the next chapters.
>
> After we learn that, we return to objects and cover them in-depth in the chapters Prototypes, inheritance and Classes.

Tasks

Two functions – one object

importance: 2

Is it possible to create functions `A` and `B` such as `new A()==new B()`?

```
function A() { ... }
function B() { ... }

let a = new A;
let b = new B;

alert( a == b ); // true
```

If it is, then provide an example of their code.

solution

Create new Calculator

importance: 5

Create a constructor function `Calculator` that creates objects with 3 methods:

- `read()` asks for two values using `prompt` and remembers them in object properties.
- `sum()` returns the sum of these properties.

- `mul()` returns the multiplication product of these properties.

For instance:

```
let calculator = new Calculator();
calculator.read();

alert( "Sum=" + calculator.sum() );
alert( "Mul=" + calculator.mul() );
```

Run the demo

Open a sandbox with tests.

solution

Create new Accumulator

importance: 5

Create a constructor function `Accumulator(startingValue)`.

Object that it creates should:

- Store the "current value" in the property `value`. The starting value is set to the argument of the constructor `startingValue`.
- The `read()` method should use `prompt` to read a new number and add it to `value`.

In other words, the `value` property is the sum of all user-entered values with the initial value `startingValue`.

Here's the demo of the code:

```
let accumulator = new Accumulator(1); // initial value 1

accumulator.read(); // adds the user-entered value
accumulator.read(); // adds the user-entered value

alert(accumulator.value); // shows the sum of these values
```

Run the demo

Open a sandbox with tests.

solution

Previous lesson Next lesson

Share Tutorial map

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 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…)