JavaScript Constructor Function

Summary: in this tutorial, you'll learn about the JavaScript constructor function and how to use the `new` keyword to create an object.

Introduction to JavaScript constructor functions

In the JavaScript objects tutorial, you learned how to use the object literal syntax to create a new object.

For example, the following creates a new person object with two properties `firstName` and `lastName` :

```
let person = {
    firstName: 'John',
    lastName: 'Doe'
};
```

In practice, you often need to create many similar objects like the `person` object.

To do that, you can use a constructor function to define a custom type and the `new` operator to create multiple objects from this type.

Technically speaking, a constructor function is a regular function with the following convention:

- The name of a constructor function starts with a capital letter like `Person` , `Document` , etc.
- A constructor function should be called only with the `new` operator.

> Note that ES6 introduces the `class` keyword that allows you to define a custom type. Classes are just syntactic sugar over the constructor functions with some enhancements.

The following example defines a constructor function called `Person` :

```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}
```

In this example, the `Person` is the same as a regular function except that its name starts with the capital letter `P` .

To create a new instance of the `Person` , you use the `new` operator:

```
let person = new Person('John','Doe');
```

Basically, the `new` operator does the following:

- Create a new empty object and assign it to the `this` variable.
- Assign the arguments `'John'` and `'Doe'` to the `firstName` and `lastName` properties of the object.
- Return the `this` value.

It's functionally equivalent to the following:

```
function Person(firstName, lastName) {
    // this = {};

    // add properties to this
    this.firstName = firstName;
    this.lastName = lastName;
    // return this;
}
```

Therefore, the following statement:

```
let person = new Person('John','Doe');
```

... returns the same result as the following statement:

```
let person = {
    firstName: 'John',
    lastName: 'Doe'
};
```

However, the constructor function `Person` allows you to create multiple similar objects. For example:

```
let person1 = new Person('Jane','Doe')
let person2 = new Person('James','Smith')
```

Adding methods to JavaScript constructor functions

An object may have methods that manipulate its data. To add a method to an object created via the constructor function, you can use the `this` keyword. For example:

```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;

    this.getFullName = function () {
        return this.firstName + " " + this.lastName;
    };
}
```

Now, you can create a new `Person` object and invoke the `getFullName()` method:

```
let person = new Person("John", "Doe");
console.log(person.getFullName());
```

Output:

```
John Doe
```

The problem with the constructor function is that when you create multiple instances of the `Person` , the `this.getFullName()` is duplicated in every instance, which is not memory efficient.

To resolve this, you can use the prototype so that all instances of a custom type can share the same methods.

Returning from constructor functions

Typically, a constructor function implicitly returns `this` that set to the newly created object. But if it has a `return` statement, then here are the rules:

- If `return` is called with an object, the constructor function returns that object instead of `this` .
- If `return` is called with a value other than an object, it is ignored.

Calling a constructor function without the new keyword

Technically, you can call a constructor function like a regular function without using the `new` keyword like this:

```
let person = Person('John','Doe');
```

In this case, the `Person` just executes like a regular function. Therefore, the `this` inside the `Person` function doesn't bind to the `person` variable but the global object.

If you attempt to access the `firstName` or `lastName` property, you'll get an error:

```
console.log(person.firstName);
```

Error:

```
TypeError: Cannot read property 'firstName' of undefined
```

Similarly, you cannot access the `getFullName()` method since it's bound to the global object.

```
person.getFullName();
```

Error:

```
TypeError: Cannot read property 'getFullName' of undefined
```

To prevent a constructor function from being invoked without the `new` keyword, ES6 introduced the `new.target` property.

If a constructor function is called with the `new` keyword, the `new.target` returns a reference of the function. Otherwise, it returns `undefined` .

The following adds a statement to the `Person` function to show the `new.target` to the console:

```
function Person(firstName, lastName) {
    console.log(new.target);

    this.firstName = firstName;
    this.lastName  = lastName;

    this.getFullName = function () {
        return this.firstName + " " + this.lastName;
    };
}
```

The following returns `undefined` because the `Person` constructor function is called like a regular function:

```
let person = Person("John", "Doe");
```

Output:

```
undefined
```

However, the following returns a reference to the `Person` function because it's called the `new` keyword:

```
let person = new Person("John", "Doe");
```

Output:

```
[Function: Person]
```

By using the `new.target` , you can force the callers of the constructor function to use the `new` keyword. Otherwise, you can throw an error like this:

```
function Person(firstName, lastName) {
    if (!new.target) {
        throw Error("Cannot be called without the new keyword");
    }
    this.firstName = firstName;
    this.lastName = lastName;
}
```

Alternatively, you can make the syntax more flexible by creating a new `Person` object if the users of the constructor function don't use the `new` keyword:

```
function Person(firstName, lastName) {
    if (!new.target) {
        return new Person(firstName, lastName);
    }

    this.firstName = firstName;
    this.lastName = lastName;
}

let person = Person("John", "Doe");

console.log(person.firstName);
```

This pattern is often used in JavaScript libraries and frameworks to make the syntax more flexible.

Summary

- JavaScript constructor function is a regular function used to create multiple similar objects.