JavaScript Prototypal Inheritance

Summary: in this tutorial, you'll learn how the JavaScript prototypal inheritance works.

Introduction to JavaScript prototypal inheritance

If you've worked with other object-oriented programming languages such as Java or C++, you've been familiar with the inheritance concept.

In this programming paradigm, a class is a blueprint for creating objects. If you want a new class to reuse the functionality of an existing class, you can create a new class that extends the existing class. This is called **classical inheritance**.

JavaScript doesn't use **classical inheritance**. Instead, it uses **prototypal inheritance**.

In prototypal inheritance, an object "inherits" properties from another object via the prototype linkage.

JavaScript prototypal inheritance and __proto__

Let's take an example to make the concept clear.

The following defines a `person` object:

```
let person = {
    name: "John Doe",
    greet: function () {
        return "Hi, I'm " + this.name;
    }
};
```

In this example, the `person` object has a property and a method:

- `name` is a property that stores the person's name.
- `greet` is a method that returns a greeting as a string.

By default, the JavaScript engine provides you with a built-in `Object()` function and an anonymous object that can be referenced by the `Object.prototype` :

> Note that the circle represents a function whereas the square represents an object.

The person object has a link to the anonymous object referenced by the `Object()` function. The `[[Prototype]]` represents the linkage:

It means that the `person` object can call any methods defined in the anonymous object referenced by the `Object.prototype` .

For example, the following shows how to call the `toString()` method via the `person` object:

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

Output:

```
[object Object]
```

The `[object Object]` is the default string representation of an object.

When you call `toString()` method via `person` , the JavaScript engine cannot find it on the `person` object. Therefore, it follows the prototype chain and searches for the method in the `Object.prototype` object.

Since the JavaScript engine can find the `toString()` method in the `Object.prototype` object, it executes the `toString()` method.

To access the prototype of the `person` object, you can use the `__proto__` property as follows

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

> Note that you should never use the `__proto__` property in the production code. Please use it for demonstration purposes only.

The following shows the `person.__proto__` and `Object.prototype` references the same object:

```
console.log(person.__proto__ === Object.prototype); // true
```

The following defines the `teacher` object that has the `teach()` method:

```
let teacher = {
    teach: function (subject) {
        return "I can teach " + subject;
    }
};
```

Like the `person` object, the `teacher.__proto__` references the `Object.prototype` as illustrated in the following picture:

If you want the `teacher` object to access all methods and properties of the `person` object, you can set the prototype of `teacher` object to the `person` object like this:

```
teacher.__proto__ = person;
```

Now, the `teacher` object can access the `name` property and `greet()` method from the `person` object via the prototype chain:

```
console.log(teacher.name);
console.log(teacher.greet());
```

Output:

```
John Doe
Hi, I'm John Doe
```

When you call the `greet()` method on the `teacher` object, the JavaScript engine finds it in the `teacher` object first.

Since the JavaScript engine cannot find the method in the `teacher` object, it follows the prototype chain and searches for the method in the `person` object. Because the JavaScript engine can find the `greet()` method in the `person` object, it executes the method.

In JavaScript, we say that the `teacher` object inherits the methods and properties of the `person` object. This kind of inheritance is called **prototypal inheritance**.

A standard way to implement prototypal inheritance in ES5

ES5 provided a standard way to work with prototypal inheritance by using the `Object.create()` method.

> Note that now you should use the newer ES6 `class` and `extends` keywords to implement inheritance. It's much simpler.

The `Object.create()` method creates a new object and uses an existing object as a prototype of the new object:

```
Object.create(proto, [propertiesObject])
```

The `Object.create()` method accepts two arguments:

- The first argument (`proto`) is an object used as the prototype for the new object.
- The second argument (`propertiesObject`), if provided, is an optional object that defines additional properties for the new object.

Suppose you have a `person` object:

```
let person = {
    name: "John Doe",
    greet: function () {
        return "Hi, I'm " + this.name;
    }
};
```

The following creates an empty `teacher` object with the `__proto__` of the `person` object:

```
let teacher = Object.create(person);
```

After that, you can define properties for the `teacher` object:

```
teacher.name = 'Jane Doe';
teacher.teach = function (subject) {
        return "I can teach " + subject;
}
```

Or you can do all of these steps in one statement as follows:

```
let teacher = Object.create(person, {
    name: { value: 'John Doe' } ,
    teach: { value: function(subject) {
        return "I can teach " + subject;
    }}
});
```

ES5 also introduced the `Object.getPrototypeOf()` method that returns the prototype of an object. For example:

```
console.log(Object.getPrototypeOf(teacher) === person);
```

Output:

```
true
```

Summary

- Inheritance allows an object to use the properties and methods of another object without duplicating the code.
- JavaScript uses the prototypal inheritance.