JavaScript Prototypal Inheritance: Constructors vs. OLOO

What do "new" and "Object.create" do?

Sun-Li Beatteay Follow
May 30, 2017 · 5 min read

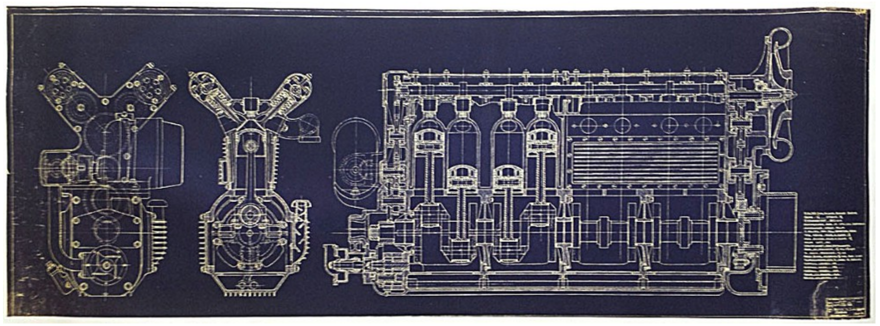
One of the first things you learn about when you study JavaScript Object Oriented Programming is Prototypal Inheritance. It can be a confusing subject to learn, especially if you're accustomed to other object oriented languages. In this piece, I'm going to focus on constructors and "Object Linking to Other Objects" (OLOO) prototypal inheritance and how each works behind the scenes.

However, before I go too much into constructors and `Object.create` , I'm going to give a brief (and rough) overview of prototypal inheritance just so we're all on the same page.

Prototypal Inheritance Overview

While prototypal inheritance is unique to JavaScript, it is not so special that it can't be compared to inheritance in other programming languages. In Ruby, for example, there are classes; subclasses inherit from superclasses. When a method is called on an object, the program will first look to see if that object has the method that is being called. If it does not, the program will then look up into it's ancestry to see if any of the classes it inherits from has that method. The program will either find the method and call it or not find it and throw an error.

JavaScript works in a somewhat similar fashion. Prototypes can be thought of as blueprints that detail the properties and behaviors any descendent object will inherit.



Think of JavaScript Prototypes as blueprints

When a new object is created, it is created using that blueprint. If you want to find the the blueprint from which a new object inherits, you can find it in its `__proto__` property.

```
var obj = new Object();
obj.__proto__

// returns

Object {__defineGetter__: function, __defineSetter__:
function, hasOwnProperty: function, __lookupGetter__:
function, __lookupSetter__: function…}
```

Each object in JavaScript has a `__proto__` property, or prototype chain, that the program uses to look up any methods that are called on that object. Similar to Ruby, if a method is called on an object in JavaScript, the program will first look to the object to see if it has that method. If it doesn't, it will look up into it's `__proto__` property to see if it is there.

Constructors

Constructor functions are one of the most common methods for creating objects that you will see in the wild. They are the more classical approach to OOP, using "classes" (constructors) to define a group of objects. Constructors will have a structure similar to this:

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

  this.fullName = function() {
    console.log(this.firstName + ' ' + this.lastName);
  }
}


var sunny = new Person('Sun-Li', 'Beatteay');


sunny.fullName()       // 'Sun-Li Beatteay'
```

So what exactly is going on here? How does `new Person()` actually create a new object and how does it affect prototypal inheritance?

When `new Person` is invoked, a new object is created and the value of `this` is set to the object. Additionally, the `constructor` property is changed to the constructor function and `__proto__` is set to the constructor's prototype. If there is no return value set at the end of the constructor function, the function will return `this`, which is the object itself.

In other words, you can rewrite,

```
var sunny = new Person('Sun-Li', 'Beatteay');
```

to this:

```
var sunny = {
  firstName: 'Sunny',
  lastName: 'Beatteay',
  fullName: function() {
    console.log(this.firstName + ' ' + this.lastName);
  },


  constructor: Person,
  __proto__: Person.prototype
};
```

In regards to prototypal inheritance, `sunny` is an instance of a `Person` object and `Person` is now in the prototype chain for `sunny`.

```
sunny instanceof Person       // true
sunny.__proto__               // Object {constructor: 
function...}
sunny.__proto__ === Person.prototype   // true
```

OLOO

The OLOO style of object creation strips away the "class" focus of object oriented programming and embraces JavaScript's prototype feature. In OLOO, objects inherit directly from other objects without needing to use a constructor as a middleman.

OLOO takes advantage of the `Object.create` method to take care of object creation and inheritance:

```
var Person = {
  init: function(first, last) {
    this.firstName = first;
    this.lastName = last;


    return this;
  },
  fullName: function() {
    console.log(this.firstName + ' ' + this.lastName);
  }
}


var sunny = Object.create(Person).init('Sunny', 'Beatteay');
sunny.fullName()    // 'Sun-Li Beatteay'
```

But what does `Object.create` do exactly? How is it different from constructors?

`Object.create` is similar to a constructor in that it creates a new object that inherits from the object passed in. In other words, it changes the value of `__proto__` on the new object to the passed in object.

However, the advantage of `Object.create` comes from it's emphasis on prototypes and delegation.

To get a better idea of what `Object.create` does, you can rewrite it as a new function:

```
function createObject(obj) {
  var newObj = {};

  newObj.__proto__ = obj;
  return newObj;
}
```

As you can see in the above function, the new object that is created doesn't have any properties or behaviors of its own. It inherits all of them from the object passed in.

If a method is called on the new object, the program will find the method in the `__proto__` property instead of in the object itself. However, if you do want to set unique properties for this object, you will need to create an initialize method like I did with `init` .

One drawback of `Object.create` is that it doesn't allow for the use of `instanceof` since it doesn't deal with the `constructor` property. Instead, to check for inheritance, you have to use the `.isPrototypeOf` method on the original object.

Example:

```
var sunny = Object.create(Person).init('Sun-Li',
'Beatteay');


Person.isPrototypeOf(sunny);    // true
```

Delegation

To be fair to constructors, delegation isn't unique to OLOO. You can separate common behaviors from constructor functions using the `prototype` property:

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


Person.prototype.fullName = function() {
  console.log(this.firstName + ' ' + this.lastName);
}
```

In the code above, if a new Person object is created, it will not contain a `fullName` method; the program will have to find it in `__proto__`.

You can even go so far as to put common properties in the prototype as well:

```
function Person() {}
```

```
Person.prototype.firstName = 'Sun-Li';

Person.prototype.lastName = 'Beatteay';

Person.prototype.fullName = function() {
  console.log(this.firstName + ' ' + this.lastName);
};
```

You can then override those properties if need be.

```
var sunny = new Person();
var iris = new Person();

iris.firstName = 'Iris';
iris.lastName = 'Sprague';

iris.fullName()   // Iris Sprague
```

Which to use?

It all comes down to personal preference. Neither of them offer greater functionality—just different tradeoffs.

Some people prefer OLOO because it embraces JavaScript's quirkiness and can save some lines of code. The drawback is the loss of `instanceof` for quick inheritance checks.

Other people prefer constructors because of it's homage to traditional class focused OOP. The drawback to constructors is that if you forget the `new` before the constructor invocation, global variables will be created and your program won't run but no errors will be thrown. Depending on how large your program is, it can be an annoying bug to deal with.

Hopefully now you have the tools and knowledge to decide for yourself which one to use. Practice using both. But when you decide on one to use for a program, keep it consistent. Future you will appreciate it.