Learn web development

Inheritance in JavaScript

see all contributors

Previous Overview: Objects Next

With most of the gory details of OOJS now explained, this article shows how to create "child" object classes (constructors) that inherit features from their "parent" classes. In addition, we present some advice on when and where you might use OOJS.

Prerequisites:	Basic computer literacy, a basic understanding of HTML and CSS, familiarity with JavaScript basics (see First steps and Building blocks) and OOJS basics (see Introduction to objects).
Objective:	To understand how it is possible to implement inheritance in JavaScript.

Prototypal inheritance

So far we have seen some inheritance in action — we have seen how prototype chains work, and how members are inherited going up a chain. But mostly this has involved built-in browser functions. How do we create an object in JavaScript that inherits from another object?

As mentioned earlier in the course, some people think JavaScript is not a true object-oriented language. In "classic OO" languages, you tend to define class objects of some kind, and you can then simply define which classes inherit from which other classes (see C++ inheritance for some simple examples). JavaScript uses a different system — "inheriting" objects do not have functionality copied over to them, instead the functionality they inherit is linked to via the prototype chain (often referred to as **prototypal inheritance**).

Let's explore how to do this with a concrete example.

Getting started

First of all, make yourself a local copy of our oojs-class-inheritance-start.html file (see it running live also). Inside here you'll find the same `Person()` constructor example that we've been using all the way through the module, with a slight difference — we've defined only the properties inside the constructor:

```
function Person(first, last, age, gender, interests) {
  this.name = {
    first,
    last
  };
  this.age = age;
  this.gender = gender;
  this.interests = interests;
};
```

The methods are *all* defined on the constructor's prototype, for example:

```
Person.prototype.greeting = function() {
  alert('Hi! I\'m ' + this.name.first + '.');
};
```

Say we wanted to create a `Teacher` class, like the one we described in our initial object-oriented definition, which inherits all the members from `Person`, but also includes:

1. A new property, `subject` — this will contain the subject the teacher teaches.
2. An updated `greeting()` method, which sounds a bit more formal than the standard `greeting()` method — more suitable for a teacher addressing some students at school.

Defining a Teacher() constructor function

The first thing we need to do is create a `Teacher()` constructor — add the following below the existing code:

```
function Teacher(first, last, age, gender, interests, subject) {
  Person.call(this, first, last, age, gender, interests);

  this.subject = subject;
}
```

This looks similar to the Person constructor in many ways, but there is something strange here that we've not seen before — the `call()` function. This function basically allows you to call a function defined somewhere else, but in the current context. The first parameter specifies the value of `this` that you want to use when running the function, and the other parameters specify the parameters that the function should have passed to it when it runs.

> **Note**: In this case we specify the inherited properties when we create a new object instance, but note that you'll need to specify them as parameters in the constructor even if the instance doesn't require them to be specified as parameters (Maybe you've got a property that's set to a random value when the object is created, for example.)

So in this case we are effectively running the `Person()` constructor function (see above) inside the `Teacher()` constructor function, resulting in the same properties being defined inside `Teacher()`, but using the values of the parameters passed to `Teacher()`, not `Person()` (we are using simply `this` as the value of `this` passed to `call()`, meaning that `this` will be the `Teacher()` function).

The last line inside the constructor simply defines the new `subject` property that teachers are going to have, which generic people don't have.

As a note, we could have simply done this:

```
function Teacher(first, last, age, gender, interests, subject) {
  this.name = {
    first,
    last
  };
  this.age = age;
  this.gender = gender;
  this.interests = interests;
  this.subject = subject;
}
```

But this is just redefining the properties anew, not inheriting them from `Person()`, so it defeats the point of what we are trying to do. It also takes more lines of code.

Setting Teacher()'s prototype and constructor reference

All is good so far, but we have a problem. We have defined a new constructor, and it has a `prototype` property that by default just contains a reference to the constructor function itself (try entering `Teacher.prototype.constructor` into your JavaScript console at this point). We need to get `Teacher()` to inherit the methods defined on `Person()`'s prototype. So how do we do that?

1. Add the following line below your previous addition:

```
Teacher.prototype = Object.create(Person.prototype);
```

   Here our friend `create()` comes to the rescue again — in this case we are using it to create a new object based on the value of the object refered by `Person.prototype`. (The `Person()` constructor's prototype object.) We then set this new object to be the value of `Teacher.prototype`. This means that `Teacher.prototype` will now inherit all the methods available on `Person.prototype`.

2. We need to do one more thing before we move on — after adding the last line, the `Teacher()` `prototype`'s `constructor` property is now equal to `Person()`, because we just set `Teacher.prototype` to be equal to an object that is a copy of `Person.prototype`! Try saving your code, loading the page in a browser, and entering `Teacher.prototype.constructor` into the console again to verify.
3. This can become a problem, so we need to set this right — you can do so by going back to your source code and adding the following line at the bottom:

```
Teacher.prototype.constructor = Teacher;
```

4. Now if you save and refresh, entering `Teacher.prototype.constructor` should return `Teacher()`, as desired, plus we are now inheriting from `Person()`!

Giving Teacher() a new greeting() function

To finish off our code, we need to define a new `greeting()` function on the `Teacher()` constructor.

The easiest way to do this is to define it on `Teacher()`'s prototype — add the following at the bottom of your code:

```
Teacher.prototype.greeting = function() {
  var prefix;

  if (this.gender === 'male' || this.gender === 'Male' || this.gender === 'm'
    prefix = 'Mr.';
  } else if (this.gender === 'female' || this.gender === 'Female' || this.gen
    prefix = 'Mrs.';
  } else {
    prefix = 'Mx.';
  }

  alert('Hello. My name is ' + prefix + ' ' + this.name.last + ', and I teach
};
```

This alerts the teacher's greeting, which also uses an appropriate name prefix for their gender, worked out using a conditional statement.

Trying the example out

Now you've entered all the code, try creating an object instance from `Teacher()` by putting the following at the bottom of your JavaScript (or something similar of your choosing):

```
var teacher1 = new Teacher('Dave', 'Griffiths', 31, 'male', ['football', 'coo
```

Now save and refresh, and try accessing the properties and methods of your new `teacher1` object, for example:

```
teacher1.name.first;
teacher1.interests[0];
teacher1.bio();
teacher1.subject;
teacher1.greeting();
```

These should all work just fine; the first three access members that were inherited from the generic `Person()` constructor (class), while the last two access members that are only available on the more specialized `Teacher()` constructor (class).

> **Note**: If you have trouble getting this to work, compare your code to our finished version (see it running live also).

The technique we covered here is not the only way to create inheriting classes in JavaScript, but it works OK, and it gives you a good idea about how to implement inheritance in JavaScript.

You might also be interested in checking out some of the new ECMAScript features that allow us to do inheritance more cleanly in JavaScript (see Classes). We didn't cover those here, as they are not yet supported very widely across browsers. All the other code constructs we discussed in this set of articles are supported as far back as IE9 or earlier, and there are ways to achieve earlier support than that.

A common way is to use a JavaScript library — most of the popular options have an easy set of functionality available for doing inheritance more easily and quickly. CoffeeScript for example provides `class`, `extends`, etc.

A further exercise

In our OOP theory section, we also included a `Student` class as a concept, which inherits all the features

of `Person`, and also has a different `greeting()` method to `Person` that is much more informal than the `Teacher`'s greeting. Have a look at what the student's greeting looks like in that section, and try implementing your own `Student()` constructor that inherits all the features of `Person()`, and implements the different `greeting()` function.

> **Note**: If you have trouble getting this to work, have a look at our finished version (see it running live also).

Object member summary

To summarize, you've basically got three types of property/method to worry about:

1. Those defined inside a constructor function that are given to object instances. These are fairly easy to spot — in your own custom code, they are the members defined inside a constructor using the `this.x = x` type lines; in built in browser code, they are the members only available to object instances (usually created by calling a constructor using the `new` keyword, e.g. `var myInstance = new myConstructor()`).
2. Those defined directly on the constructor themselves, that are available only on the constructor. These are commonly only available on built-in browser objects, and are recognized by being chained directly onto a constructor, *not* an instance. For example, `Object.keys()`.
3. Those defined on a constructor's prototype, which are inherited by all instances and inheriting object classes. These include any member defined on a Constructor's prototype property, e.g. `myConstructor.prototype.x()`.

If you are not sure which is which, don't worry about it just yet — you are still learning, and familiarity will come with practice.

When would you use inheritance in JavaScript?

Particularly after this last article, you might be thinking "woo, this is complicated". Well, you are right, prototypes and inheritance represent some of the most complex aspects of JavaScript, but a lot of JavaScript's power and flexibility comes from its object structure and inheritance, and it is worth understanding how it works.

In a way, you use inheritance all the time — whenever you use various features of a WebAPI , or methods/properties defined on a built-in browser object that you call on your strings, arrays, etc., you are implicitly using inheritance.

In terms of using inheritance in your own code, you probably won't use it that often, especially to begin with, and in small projects — it is a waste of time to use objects and inheritance just for the sake of it, when you don't need them. But as your code bases get larger, you are more likely to find a need for it. If you find yourself starting to create a number of objects that have similar features, then creating a generic object type to contain all the shared functionality and inheriting those features in more specialized object types can be convenient and useful.

> **Note**: Because of the way JavaScript works, with the prototype chain, etc., the sharing of functionality between objects is often called **delegation** — the specialized objects delegate that functionality to the generic object type. This is probably more accurate than calling it *inheritance*, as the "inherited" functionality is not copied to the objects that are doing the "inheriting". Instead it still remains in the generic object.

When using inheritance, you are advised to not have too many levels of inheritance, and to keep careful track of where you define your methods and properties. It is possible to start writing code that temporarily modifies the prototypes of built-in browser objects, but you should not do this unless you have a really good reason. Too much inheritance can lead to endless confusion, and endless pain when you try to debug such code.

Ultimately, objects are just another form of code reuse, like functions or loops, with their own specific roles and advantages. If you find yourself creating a bunch of related variables and functions and want to track them all together and package them neatly, an object is a good idea. Objects are also very useful when you want to pass a collection of data from one place to another. Both of these things can be achieved without use of constructors or inheritance. If you only need a single instance of an object, then you are probably better off just using an object literal, and you certainly don't need inheritance.

Summary

This article has covered the remainder of the core OOJS theory and syntax that we think you should know now. At this point you should understand JavaScript object and OOP basics, prototypes and prototypal inheritance, how to create classes (constructors) and object instances, add features to classes, and create subclasses that inherit from other classes.

In the next article we'll have a look at how to work with JavaScript Object Notation (JSON), a common data exchange format written using JavaScript objects.

See also

- ObjectPlayground.com — A really useful interactive learning site for learning about objects.
- Secrets of the JavaScript Ninja, Chapter 6 — A good book on advanced JavaScript concepts and techniques, by John Resig and Bear Bibeault. Chapter 6 covers aspects of prototypes and inheritance really well; you can probably track down a print or online copy fairly easily.
- You Don't Know JS: this & Object Prototypes — Part of Kyle Simpson's excellent series of JavaScript manuals, Chapter 5 in particular looks at prototypes in much more detail than we do here. We've presented a simplified view in this series of articles aimed at beginners, whereas Kyle goes into great depth and provides a more complex but more accurate picture.

← Previous ↑ Overview: Objects Next →

Was this article helpful?