Inheritance in JavaScript

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, and look at how classes are dealt with in modern ECMAScript syntax.

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?

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 + '.');
};
```

Note: In the source code, you'll also see `bio()` and `farewell()` methods defined. Later you'll see how these can be inherited by other constructors.

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 are those that should be passed to the function when it is invoked.

We want the `Teacher()` constructor to take the same parameters as the `Person()` constructor it is inheriting from, so we specify them all as parameters in the `call()` invocation.

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.

Inheriting from a constructor with no parameters

Note that if the constructor you are inheriting from doesn't take its property values from parameters, you don't need to specify them as additional arguments in `call()`. So, for example, if you had something really simple like this:

```
function Brick() {
  this.width = 10;
  this.height = 20;
}
```

You could inherit the `width` and `height` properties by doing this (as well as the other steps described below, of course):

```
function BlueGlassBrick() {
  Brick.call(this);

  this.opacity = 0.5;
  this.color = 'blue';
}
```

Note that we've only specified `this` inside `call()` — no other parameters are required as we are not inheriting any properties from the parent that are set via parameters.

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, which by default just contains an object with a reference to the constructor function itself. It does not contain the methods of the Person constructor's `prototype` property. To see this, enter `Object.getOwnPropertyNames(Teacher.prototype)` into either the text input field or your JavaScript console. Then enter it again, replacing `Teacher` with `Person`. Nor does the new constructor *inherit* those methods. To see this, compare the outputs of `Person.prototype.greeting` and `Teacher.prototype.greeting`. 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 and make it the value of `Teacher.prototype`. The new object has `Person.prototype` as its prototype and will therefore inherit, if and when needed, all the methods available on `Person.prototype`.
2. We need to do one more thing before we move on. After adding the last line, `Teacher.` `prototype`'s `constructor` property is now equal to `Person()`, because we just set `Teacher.prototype` to reference an object that inherits its properties from `Person.prototype`! Try saving your code, loading the page in a browser, and entering `Teacher.prototype.constructor` into the console 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:

   ```
   Object.defineProperty(Teacher.prototype, 'constructor', {
       value: Teacher,
       enumerable: false, // so that it does not appear in 'for in' loop
       writable: true });
   ```

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() {
  let prefix;

  if (this.gender === 'male' || this.gender === 'Male' || this.gender === 'm' || t
    prefix = 'Mr.';
  } else if (this.gender === 'female' || this.gender === 'Female' || this.gender =
    prefix = 'Ms.';
  } 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 that 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):

```
let teacher1 = new Teacher('Dave', 'Griffiths', 31, 'male', ['football', 'cookery'
```

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

```
1  teacher1.name.first;
2  teacher1.interests[0];
3  teacher1.bio();
4  teacher1.subject;
5  teacher1.greeting();
6  teacher1.farewell();
```

These should all work just fine. The queries on lines 1, 2, 3, and 6 access members inherited from the generic `Person()` constructor (class). The query on line 4 accesses a member that is available only on the more specialized `Teacher()` constructor (class). The query on line 5 would have accessed a member inherited from `Person()`, except for the fact that `Teacher()` has its own member with the same name, so the query accesses that member.

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 from `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 got four 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. `let 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()`. These are also known as **static properties/methods**.
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()`.
4. Those available on an object instance, which can either be an object created when a constructor is instantiated like we saw above (so for example `var teacher1 = new Teacher( name = 'Chris' );` and then `teacher1.name`), or an object literal (`let teacher1 = { name = 'Chris' }` and then `teacher1.name`).

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.

ECMAScript 2015 Classes

ECMAScript 2015 introduces class syntax to JavaScript as a way to write reusable classes using easier, cleaner syntax, which is more similar to classes in C++ or Java. In this section we'll convert the Person and Teacher examples from prototypal inheritance to classes, to show you how it's done.

Note: This modern way of writing classes is supported in all modern browsers, but it is still worth knowing about the underlying prototypal inheritance in case you work on a project that requires supporting a browser that doesn't support this syntax (most notably Internet Explorer).

Let's look at a rewritten version of the Person example, class-style:

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

  greeting() {
    console.log(`Hi! I'm ${this.name.first}`);
  };

  farewell() {
    console.log(`${this.name.first} has left the building. Bye for now!`);
  };
}
```

The class statement indicates that we are creating a new class. Inside this block, we define all the features of the class:

- The `constructor()` method defines the constructor function that represents our `Person` class.
- `greeting()` and `farewell()` are class methods. Any methods you want associated with the class are defined inside it, after the constructor. In this example, we've used template literals rather than string concatenation to make the code easier to read.

We can now instantiate object instances using the `new` operator, in just the same way as we did before:

```
let han = new Person('Han', 'Solo', 25, 'male', ['Smuggling']);
han.greeting();
// Hi! I'm Han

let leia = new Person('Leia', 'Organa', 19, 'female', ['Government']);
leia.farewell();
// Leia has left the building. Bye for now
```

Note: Under the hood, your classes are being converted into Prototypal Inheritance models — this is just syntactic sugar. But I'm sure you'll agree that it's easier to write.

Inheritance with class syntax

Above we created a class to represent a person. They have a series of attributes that are common to all people; in this section we'll create our specialized `Teacher` class, making it inherit from `Person` using modern class syntax. This is called creating a subclass or subclassing.

To create a subclass we use the extends keyword to tell JavaScript the class we want to base our class on,

```
class Teacher extends Person {
  constructor(subject, grade) {
    this.subject = subject;
    this.grade = grade;
  }
}
```

but there's a little catch.

Unlike old-school constructor functions where the `new` operator does the initialization of `this` to a newly-allocated object, this isn't automatically initialized for a class defined by the extends keyword, i.e the sub-classes.

Therefore running the above code will give an error:

```
Uncaught ReferenceError: Must call super constructor in derived class before
accessing 'this' or returning from derived constructor
```

For sub-classes, the `this` intialization to a newly allocated object is always dependant on the parent class constructor, i.e the constructor function of the class from which you're extending.

Here we are extending the `Person` class — the `Teacher` sub-class is an extension of the `Person` class. So for `Teacher`, the `this` initialization is done by the `Person` constructor.

To call the parent constructor we have to use the `super()` operator, like so:

```
class Teacher extends Person {
  constructor(subject, grade) {
    super(); // Now 'this' is initialized by calling the parent constructor.
    this.subject = subject;
    this.grade = grade;
  }
}
```

There is no point having a sub-class if it doesn't inherit properties from the parent class.
It is good then, that the `super()` operator also accepts arguments for the parent constructor.

Looking back to our `Person` constructor, we can see it has the following block of code in its constructor method:

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

Since the `super()` operator is actually the parent class constructor, passing it the necessary arguments of the `Parent` class constructor will also initialize the parent class properties in our sub-class, thereby inheriting it:

```
class Teacher extends Person {
  constructor(first, last, age, gender, interests, subject, grade) {
    super(first, last, age, gender, interests);

    // subject and grade are specific to Teacher
    this.subject = subject;
    this.grade = grade;
  }
}
```

Now when we instantiate `Teacher` object instances, we can call methods and properties defined on both `Teacher` and `Person` as we'd expect:

```
let snape = new Teacher('Severus', 'Snape', 58, 'male', ['Potions'], 'Dark arts',
snape.greeting(); // Hi! I'm Severus.
snape.farewell(); // Severus has left the building. Bye for now.
snape.age // 58
snape.subject; // Dark arts
```

Like we did with Teachers, we could create other subclasses of `Person` to make them more specialized without modifying the base class.

Note: You can find this example on GitHub as es2015-class-inheritance.html (see it live also).

Getters and Setters

There may be times when we want to change the values of an attribute in the classes we create or we don't know what the final value of an attribute will be. Using the `Teacher` example, we may not know what subject the teacher will teach before we create them, or their subject may change between terms.

We can handle such situations with getters and setters.

Let's enhance the Teacher class with getters and setters. The class starts the same as it was the last time we looked at it.

Getters and setters work in pairs. A getter returns the current value of the variable and its corresponding setter changes the value of the variable to the one it defines.

The modified `Teacher` class looks like this:

```
class Teacher extends Person {
  constructor(first, last, age, gender, interests, subject, grade) {
    super(first, last, age, gender, interests);
    // subject and grade are specific to Teacher
    this._subject = subject;
    this.grade = grade;
  }

  get subject() {
    return this._subject;
  }

  set subject(newSubject) {
    this._subject = newSubject;
  }
}
```

In our class above we have a getter and setter for the `subject` property. We use `_` to create a separate value in which to store our name property. Without using this convention, we would get errors every time we called get or set. At this point:

- To show the current value of the `_subject` property of the `snape` object we can use the `snape.subject` getter method.
- To assign a new value to the `_subject` property we can use the `snape.subject="new value"` setter method.

The example below shows the two features in action:

```
// Check the default value
console.log(snape.subject) // Returns "Dark arts"

// Change the value
snape.subject = "Balloon animals" // Sets _subject to "Balloon animals"

// Check it again and see if it matches the new value
console.log(snape.subject) // Returns "Balloon animals"
```

Note: You can find this example on GitHub as es2015-getters-setters.html (see it live also).

Note: Getters and setters can be very useful at times, for example when you want to run some code every time a property is requested or set. For simple cases, however, plain property access without a getter or setter will do just fine.

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 Web API , 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 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**. Specialized objects delegate functionality to a generic object type.

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.

Alternatives for extending the prototype chain

In JavaScript, there are several different ways to extend the prototype of an object aside from what we've shown above. To find out more about the other ways, visit our Inheritance and the prototype chain article.

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 7 — A good book on advanced JavaScript concepts and techniques, by John Resig, Bear Bibeault, and Josip Maras. Chapter 7 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 →

In this module

- Object basics
- Object-oriented JavaScript for beginners
- Object prototypes
- Inheritance in JavaScript

- Working with JSON data
- Object building practice
- Adding features to our bouncing balls demo

Last modified: Dec 28, 2019, by MDN contributors

Related Topics