Learn web development

Object prototypes

see all contributors

Prototypes are the mechanism by which JavaScript objects inherit features from one another, and they work differently than inheritance mechanisms in classical object-oriented programming languages. In this article we explore that difference, explain how prototype chains work, and look at how the prototype property can be used to add methods to existing constructors.

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 JavaScript object prototypes, how prototype chains work, and how to add new methods onto the prototype property.

A prototype-based language?

JavaScript is often described as a **prototype-based language** — each object has a **prototype object**, which acts as a template object that it inherits methods and properties from. An object's prototype object may also have a prototype object, which it inherits methods and properties from, and so on. This is often referred to as a **prototype chain**, and explains why different objects have properties and methods defined on other objects available to them.

Well, to be exact, the properties and methods are defined on the Objects' constructor functions, not the object instances themselves.

In classic OOP, classes are defined, then when object instances are created all the properties and methods defined on the class are copied over to the instance. In JavaScript, they are not copied over —

instead, a link is made between the object instance and its constructor (a link in the prototype chain), and the properties and methods are found in the constructor by walking up the chain.

Let's look at an example to make this a bit clearer.

Understanding prototype objects

Let's go back to the example in which we finished writing our `Person()` constructor — load the example in your browser. If you don't still have it from working through the last article, use our oojs-class-further-exercises.html example (see also the source code).

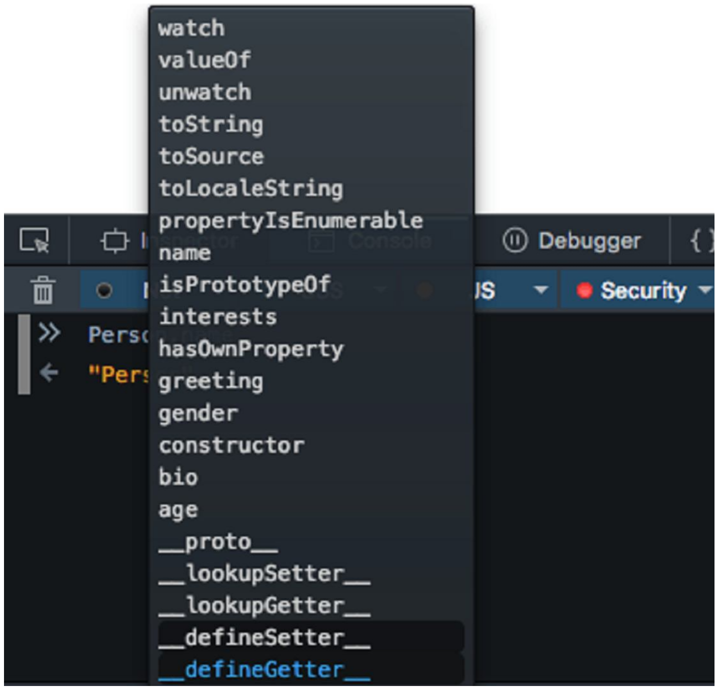
In this example we have defined a constructor function, like so:

```
function Person(first, last, age, gender, interests) {

  // property and method definitions

};
```

We have then created an object instance like this:

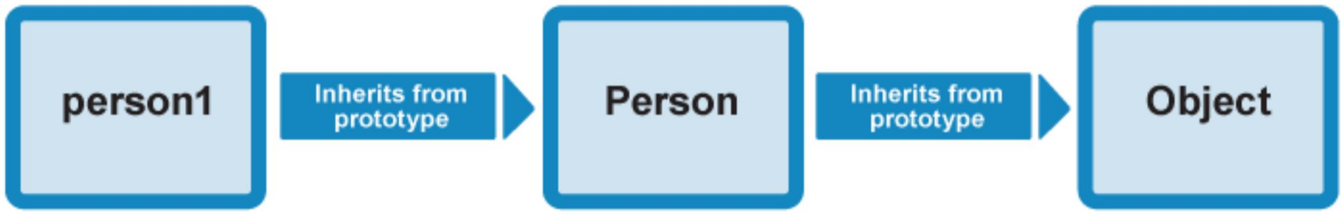
```
var person1 = new Person('Bob', 'Smith', 32, 'male', ['music', 'skiing']);
```

If you type "`person1.`" into your JavaScript console, you should see the browser try to auto-complete this with the member names available on this object:


```
watch
valueOf
unwatch
toString
toSource
toLocaleString
propertyIsEnumerable
name
isPrototypeOf
interests
hasOwnProperty
greeting
gender
constructor
bio
age
__proto__
__lookupSetter__
__lookupGetter__
__defineSetter__
__defineGetter__
```

```
» person1.__defineGetter__
```

In this list you will see the members defined on `person1`'s prototype object, which is the `Person()` constructor — `name`, `age`, `gender`, `interests`, `bio`, and `greeting`. You will however also see some other members — `watch`, `valueOf`, etc — these are defined on the `Person()` constructor's prototype object, which is `Object`. This demonstrates the prototype chain working.



So what happens if you call a method on `person1`, which is actually defined on `Object`? For example:

```
person1.valueOf()
```

This method simply returns the value of the object it is called on — try it and see! In this case, what happens is:

- The browser initially checks to see if the `person1` object has a `valueOf()` method available on it.
- It doesn't, so the browser then checks to see if the `person1` object's prototype object (`Person`) has a `valueOf()` method available on it.
- It doesn't either, so the browser then checks to see if the `Person()` constructor's prototype object (`Object`) has a `valueOf()` method available on it. It does, so it is called, and all is good!

Note: We want to reiterate that the methods and properties are **not** copied from one object to another in the prototype chain — they are accessed by walking up the chain as described above.

Note: There isn't officially a way to access an object's prototype object directly — the "links" between the items in the chain are defined in an internal property, referred to as `[[prototype]]` in the specification for the JavaScript language (see ECMAScript). Most modern browsers however do have a property available on them called `__proto__` (that's 2 underscores either side), which contains the object's prototype object. For example, try `person1.__proto__` and `person1.__proto__.__proto__` to see what the chain looks like in code!

The prototype property: Where inherited members are defined

So, where are the inherited properties and methods defined? If you look at the `Object` reference page, you'll see listed in the left hand side a large number of properties and methods — many more than the number of inherited members we saw available on the `person1` object in the above screenshot. Some are inherited, and some aren't — why is this?

The answer is that the inherited ones are the ones defined on the `prototype` property (you could call it a sub namespace) — that is, the ones that begin with `Object.prototype.`, and not the ones that begin with just `Object.` The `prototype` property's value is an object, which is basically a bucket for storing properties and methods that we want to be inherited by objects further down the prototype chain.

So `Object.prototype.watch()`, `Object.prototype.valueOf()`, etc., are available to any object types that inherit from `Object.prototype`, including new object instances created from the constructor.

`Object.is()`, `Object.keys()`, and other members not defined inside the `prototype` bucket are not inherited by object instances or object types that inherit from `Object.prototype`. They are methods/properties available just on the `Object()` constructor itself.

> **Note**: This seems strange — how can you have a method defined on a constructor, which is itself a function? Well, a function is also a type of object — see the `Function()` constructor reference if you don't believe us.

1. You can check out existing prototype properties for yourself — go back to our previous example and try entering the following into the JavaScript console:

   ```
   Person.prototype
   ```

2. The output won't show you very much — after all, we haven't defined anything on our custom constructor's prototype! By default, a constructor's `prototype` always starts empty. Now try the following:

   ```
   Object.prototype
   ```

You'll see a large number of methods defined on `Object`'s `prototype` property, which are then available on objects that inherit from `Object`, as shown earlier.

You'll see other examples of prototype chain inheritance all over JavaScript — try looking for the methods and properties defined on the prototype of the `String`, `Date`, `Number`, and `Array` global objects, for example. These all have a number of members defined on their prototype, which is why for example when you create a string, like this:

```
var myString = 'This is my string.';
```

`myString` immediately has a number of useful methods available on it, like `split()`, `indexOf()`, `replace()`, etc.

> **Important**: The `prototype` property is one of the most confusingly-named parts of JavaScript — you might think that this points to the prototype object of the current object, but it doesn't (that's an internal object that can be accessed by `__proto__`, remember?). `prototype` instead is a property containing an object on which you define members that you want to be inherited.

Revisiting create()

Earlier on we showed how the `Object.create()` method can be used to create a new object instance.

1. For example, try this in your previous example's JavaScript console:
   ```
   var person2 = Object.create(person1);
   ```
2. What `create()` actually does is to create a new object from a specified prototype object. Here, `person2` is being created using `person1` as a prototype object. You can check this by entering the following in the console:
   ```
   person2.__proto__
   ```

This will return the `person1` object.

The constructor property

Every function has a prototype property whose value is an object containing a `constructor` property. This constructor property points to the original constructor function. As you will see in the next section that properties defined on the Person.prototype property (or in general on a constructor function's prototype property which is an object as mentioned in the above section) become available to all the instance objects created using the Person() constructor. Hence, the constructor property is also available to both person1 and person2 objects.

1. For example, try these commands in the console:
   ```
   person1.constructor
   person2.constructor
   ```

   These should both return the `Person()` constructor, as it contains the original definition of these instances.

   A clever trick is that you can put parentheses onto the end of the `constructor` property (containing any required parameters) to create another object instance from that constructor. The constructor is a function after all, so can be invoked using parentheses; you just need to include the `new` keyword to specify that you want to use the function as a constructor.

2. Try this in the console:

```
var person3 = new person1.constructor('Karen', 'Stephenson', 26, 'female
```

3. Now try accessing your new object's features, for example:

```
person3.name.first
person3.age
person3.bio()
```

This works well. You won't need to use it often, but it can be really useful when you want to create a new instance and don't have a reference to the original constructor easily available for some reason.

The `constructor` property has other uses besides. For example, if you have an object instance and you want to return the name of the constructor it is an instance of, you can use the following:

```
instanceName.constructor.name
```

Try this, for example:

```
person1.constructor.name
```

Modifying prototypes

Let's have a look at an example of modifying the `prototype` property of a constructor function (methods added to the prototype are then available on all object instances created from the constructor).

1. Go back to our oojs-class-further-exercises.html example and make a local copy of the source code. Below the existing JavaScript, add the following code, which adds a new method to the constructor's `prototype` property:

```
Person.prototype.farewell = function() {
  alert(this.name.first + ' has left the building. Bye for now!');
}
```

2. Save the code and load the page in the browser, and try entering the following into the text input:

```
person1.farewell();
```

You should get an alert message displayed, featuring the person's name as defined inside the constructor. This is really useful, but what is even more useful is that the whole inheritance chain has updated dynamically, automatically making this new method available on all object instances derived from the constructor.

Think about this for a moment. In our code we define the constructor, then we create an instance object from the constructor, *then* we add a new method to the constructor's prototype:

```
function Person(first, last, age, gender, interests) {

  // property and method definitions

};

var person1 = new Person('Tammi', 'Smith', 32, 'neutral', ['music', 'skiing',

Person.prototype.farewell = function() {
  alert(this.name.first + ' has left the building. Bye for now!');
}
```

But the `farewell()` method is *still* available on the `person1` object instance — its available functionality has been automatically updated. This proves what we said earlier about the prototype chain, and the browser looking upwards in the chain to find methods that aren't defined on the object instance itself rather than those methods being copied to the instance. This provides a very powerful, extensible system of functionality.

Note: If you are having trouble getting this example to work, have a look at our oojs-class-prototype.html example (see it running live also).

You will rarely see properties defined on the `prototype` property, because they are not very flexible when defined like this. For example you could add a property like so:

```
Person.prototype.fullName = 'Bob Smith';
```

But this isn't very flexible, as the person might not be called that. It'd be much better to do this, to build the `fullName` out of `name.first` and `name.last`:

```
Person.prototype.fullName = this.name.first + ' ' + this.name.last;
```

This however doesn't work, as `this` will be referencing the global scope in this case, not the function scope. Calling this property would return `undefined undefined`. This worked fine on the method we defined earlier in the prototype because it is sitting inside a function scope, which will be transferred successfully to the object instance scope. So you might define constant properties on the prototype (i.e. ones that never need to change), but generally it works better to define properties inside the constructor.

In fact, a fairly common pattern for more object definitions is to define the properties inside the constructor, and the methods on the prototype. This makes the code easier to read, as the constructor only contains the property definitions, and the methods are split off into separate blocks. For example:

```
// Constructor with property definitions

function Test(a, b, c, d) {
  // property definitions
};

// First method definition

Test.prototype.x = function() { ... }

// Second method definition

Test.prototype.y = function() { ... }

// etc.
```

This pattern can be seen in action in Piotr Zalewa's school plan app example.

Summary

This article has covered JavaScript object prototypes, including how prototype object chains allow objects to inherit features from one another, the prototype property and how it can be used to add methods to constructors, and other related topics.

In the next article we'll look at how you can implement inheritance of functionality between two of your own custom objects.

Previous Overview: Objects Next