Object prototypes

Prototypes are the mechanism by which JavaScript objects inherit features from one another. In this article, we explain what a prototype is, how prototype chains work, and how a prototype for an object can be set.

Prerequisites:	Understanding JavaScript functions, familiarity with JavaScript basics (see <u>First steps</u> and <u>Building blocks</u>), and OOJS basics (see <u>Introduction to objects</u>).
Objective:	To understand JavaScript object prototypes, how prototype chains work, and how to set the prototype of an object.

The prototype chain

In the browser's console, try creating an object literal:

```
const myObject = {
  city: "Madrid",
  greet() {
    console.log(`Greetings from ${this.city}`);
  },
};

myObject.greet(); // Greetings from Madrid
```

This is an object with one data property, `city`, and one method, `greet()`. If you type the object's name *followed by a period* into the console, like `myObject.`, then the console will pop up a list of all the properties available to this object. You'll see that as well as `city` and `greet`, there are lots of other properties!

```
__defineGetter__
__defineSetter__
__lookupGetter__
__lookupSetter__
__proto__
city
constructor
greet
hasOwnProperty
isPrototypeOf
propertyIsEnumerable
toLocaleString
toString
valueOf
```

Try accessing one of them:

```
myObject.toString(); // "[object Object]"
```

It works (even if it's not obvious what `toString()` does).

What are these extra properties, and where do they come from?

Every object in JavaScript has a built-in property, which is called its prototype. The prototype is itself an object, so the prototype will have its own prototype, making what's called a prototype chain. The chain ends when we reach a prototype that has `null` for its own prototype.

> Note: The property of an object that points to its prototype is not called `prototype`. Its name is not standard, but in practice all browsers use <u>`__proto__`</u>. The standard way to access an object's prototype is the <u>`Object.getPrototypeOf()`</u> method.

When you try to access a property of an object: if the property can't be found in the object itself, the prototype is searched for the property. If the property still can't be found, then the prototype's prototype is searched, and so on until either the property is found, or the end of the chain is reached, in which case `undefined` is returned.

So when we call `myObject.toString()` , the browser:

- looks for `toString` in `myObject`
- can't find it there, so looks in the prototype object of `myObject` for `toString`
- finds it there, and calls it.

What is the prototype for `myObject` ? To find out, we can use the function `Object.getPrototypeOf()` :

```
Object.getPrototypeOf(myObject); // Object { }
```

This is an object called `Object.prototype` , and it is the most basic prototype, that all objects have by default. The prototype of `Object.prototype` is `null` , so it's at the end of the prototype chain:
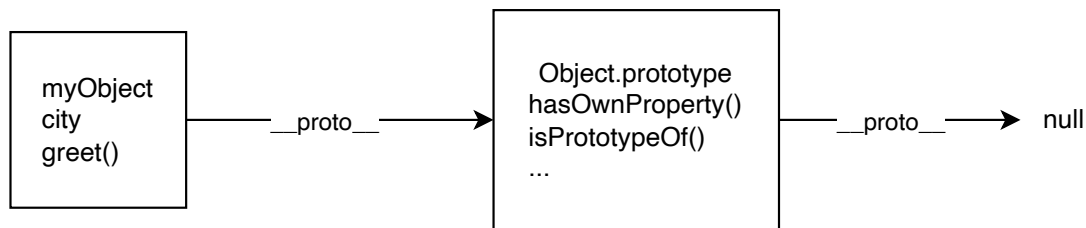
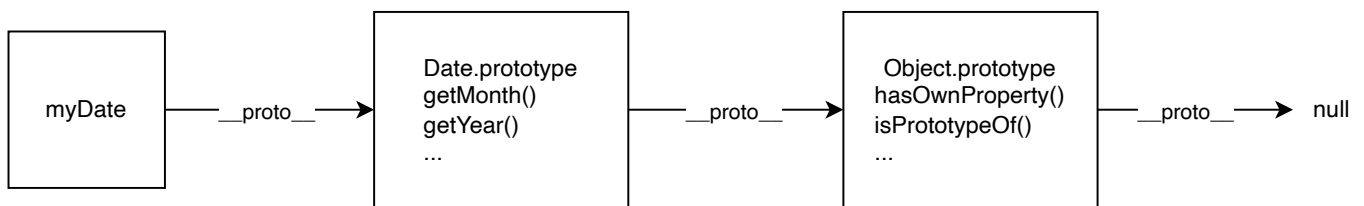Viewer does not support full SVG 1.1

The prototype of an object is not always `Object.prototype` . Try this:

```
const myDate = new Date();
let object = myDate;

do {
  object = Object.getPrototypeOf(object);
  console.log(object);
} while (object);

// Date.prototype
// Object { }
// null
```

This code creates a `Date` object, then walks up the prototype chain, logging the prototypes. It shows us that the prototype of `myDate` is a `Date.prototype` object, and the prototype of *that* is `Object.prototype` .

Viewer does not support full SVG 1.1

In fact, when you call familiar methods, like `myDate2.getMonth()` , you are calling a method that's defined on `Date.prototype` .

Shadowing properties

What happens if you define a property in an object, when a property with the same name is defined in the object's prototype? Let's see:

```
const myDate = new Date(1995, 11, 17);

console.log(myDate.getYear()); // 95

myDate.getYear = function () {
  console.log("something else!");
};

myDate.getYear(); // 'something else!'
```

This should be predictable, given the description of the prototype chain. When we call `getYear()` the browser first looks in `myDate` for a property with that name, and only checks the prototype if `myDate` does not define it. So when we add `getYear()` to `myDate`, then the version in `myDate` is called.

This is called "shadowing" the property.

Setting a prototype

There are various ways of setting an object's prototype in JavaScript, and here we'll describe two: `Object.create()` and constructors.

Using Object.create

The `Object.create()` method creates a new object and allows you to specify an object that will be used as the new object's prototype.

Here's an example:

```
const personPrototype = {
  greet() {
    console.log("hello!");
  },
};

const carl = Object.create(personPrototype);
carl.greet(); // hello!
```

Here we create an object `personPrototype`, which has a `greet()` method. We then use `Object.create()` to create a new object with `personPrototype` as its prototype. Now we can call `greet()` on the new object, and the prototype provides its implementation.

Using a constructor

In JavaScript, all functions have a property named `prototype`. When you call a function as a constructor, this property is set as the prototype of the newly constructed object (by convention, in the property named `__proto__`).

So if we set the `prototype` of a constructor, we can ensure that all objects created with that constructor are given that prototype:

```
const personPrototype = {
  greet() {
    console.log(`hello, my name is ${this.name}!`);
  },
};

function Person(name) {
  this.name = name;
}

Object.assign(Person.prototype, personPrototype);
// or
// Person.prototype.greet = personPrototype.greet;
```

Here we create:

- an object `personPrototype`, which has a `greet()` method
- a `Person()` constructor function which initializes the name of the person to create.

We then put the methods defined in `personPrototype` onto the `Person` function's `prototype` property using Object.assign.

After this code, objects created using `Person()` will get `Person.prototype` as their prototype, which automatically contains the `greet` method.

```
const reuben = new Person("Reuben");
reuben.greet(); // hello, my name is Reuben!
```

This also explains why we said earlier that the prototype of `myDate` is called `Date.prototype`: it's the `prototype` property of the `Date` constructor.

Own properties

The objects we create using the Person constructor above have two properties:

- a name property, which is set in the constructor, so it appears directly on Person objects
- a greet() method, which is set in the prototype.

It's common to see this pattern, in which methods are defined on the prototype, but data properties are defined in the constructor. That's because methods are usually the same for every object we create, while we often want each object to have its own value for its data properties (just as here where every person has a different name).

Properties that are defined directly in the object, like name here, are called own properties, and you can check whether a property is an own property using the static Object.hasOwn() method:

```
const irma = new Person("Irma");

console.log(Object.hasOwn(irma, "name")); // true
console.log(Object.hasOwn(irma, "greet")); // false
```

> Note: You can also use the non-static Object.hasOwnProperty() method here, but we recommend that you use Object.hasOwn() if you can.

Prototypes and inheritance

Prototypes are a powerful and very flexible feature of JavaScript, making it possible to reuse code and combine objects.

In particular they support a version of inheritance. Inheritance is a feature of object-oriented programming languages that lets programmers express the idea that some objects in a system are more specialized versions of other objects.

For example, if we're modeling a school, we might have *professors* and *students*: they are both *people*, so have some features in common (for example, they both have names), but each might add extra features (for example, professors have a subject that they teach), or might implement the same feature in different ways. In an OOP system we might say that professors and students both inherit from people.

You can see how in JavaScript, if Professor and Student objects can have Person prototypes, then they can inherit the common properties, while adding and redefining those properties which need to differ.

In the next article we'll discuss inheritance along with the other main features of object-oriented programming languages, and see how JavaScript supports them.

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.