Inheritance and the prototype chain

developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

JavaScript is a bit confusing for developers experienced in class-based languages (like Java or C++), as it is dynamic and does not provide a `class` implementation per se (the `class` keyword is introduced in ES2015, but is syntactical sugar, JavaScript remains prototype-based).

When it comes to inheritance, JavaScript only has one construct: objects. Each object has a private property which holds a link to another object called its **prototype**. That prototype object has a prototype of its own, and so on until an object is reached with `null` as its prototype. By definition, `null` has no prototype, and acts as the final link in this **prototype chain**.

Nearly all objects in JavaScript are instances of Object which sits on the top of a prototype chain.

While this confusion is often considered to be one of JavaScript's weaknesses, the prototypal inheritance model itself is, in fact, more powerful than the classic model. It is, for example, fairly trivial to build a classic model on top of a prototypal model.

Inheritance with the prototype chain Section

Inheriting properties Section

JavaScript objects are dynamic "bags" of properties (referred to as **own properties**). JavaScript objects have a link to a prototype object. When trying to access a property of an object, the property will not only be sought on the object but on the prototype of the object, the prototype of the prototype, and so on until either a property with a matching name is found or the end of the prototype chain is reached.

Following the ECMAScript standard, the notation `someObject.[[Prototype]]` is used to designate the prototype of `someObject`. Since ECMAScript 2015, the `[[Prototype]]` is accessed using the accessors

Object.getPrototypeOf() and Object.setPrototypeOf(). This is equivalent to the JavaScript property `__proto__` which is non-standard but de-facto implemented by many browsers.

It should not be confused with the `func.prototype` property of functions, which instead specifies the `[[Prototype]]` to be assigned to all *instances* of objects created by the given function when used as a constructor. The **`Object.prototype`** property represents the Object prototype object.

Here is what happens when trying to access a property:

```
let f = function () {
  this.a = 1;
  this.b = 2;
}
let o = new f();
```

```
f.prototype.b = 3;
f.prototype.c = 4;
```

```
console.log(o.a);
```

```
console.log(o.b);
```

```
console.log(o.c);
```

```
console.log(o.d);
```

Code Link

Setting a property to an object creates an own property. The only exception to the getting and setting behavior rules is when there is an inherited property with a getter or a setter.

Inheriting "methods" Section

JavaScript does not have "methods" in the form that class-based languages define them. In JavaScript, any function can be added to an object in the form of a property. An inherited function acts just as any other property, including property shadowing as shown above (in this case, a form of *method overriding*).

When an inherited function is executed, the value of this points to the inheriting object, not to the prototype object where the function is an own property.

```
var o = {
  a: 2,
  m: function() {
    return this.a + 1;
  }
};

console.log(o.m());


var p = Object.create(o);


p.a = 4;
console.log(p.m());
```

Using prototypes in JavaScript Section

Let's look at what happens behind the scenes in a bit more detail.

In JavaScript, as mentioned above, functions are able to have properties. All functions have a special property named `prototype`. Please note that the code below is free-standing (it is safe to assume there is no other JavaScript on the webpage other than the below code). For the best learning experience, it is highly recommended that you open a console (which, in Chrome and Firefox, can be done by pressing Ctrl+Shift+I), navigate to the "console" tab, copy-and-paste in the below JavaScript code, and run it by pressing the Enter/Return key.

```
function doSomething(){}
console.log( doSomething.prototype );
```

```
var doSomething = function(){};
console.log( doSomething.prototype );
```

As seen above, `doSomething()` has a default `prototype` property, as demonstrated by the console. After running this code, the console should have displayed an object that looks similar to this.

```
{
    constructor: ƒ doSomething(),
    __proto__: {
        constructor: ƒ Object(),
        hasOwnProperty: ƒ hasOwnProperty(),
        isPrototypeOf: ƒ isPrototypeOf(),
        propertyIsEnumerable: ƒ propertyIsEnumerable(),
        toLocaleString: ƒ toLocaleString(),
        toString: ƒ toString(),
        valueOf: ƒ valueOf()
    }
}
```

We can add properties to the prototype of `doSomething()`, as shown below.

```
function doSomething(){}
doSomething.prototype.foo = "bar";
console.log( doSomething.prototype );
```

This results in:

```
{
    foo: "bar",
    constructor: ƒ doSomething(),
    __proto__: {
        constructor: ƒ Object(),
        hasOwnProperty: ƒ hasOwnProperty(),
        isPrototypeOf: ƒ isPrototypeOf(),
        propertyIsEnumerable: ƒ propertyIsEnumerable(),
        toLocaleString: ƒ toLocaleString(),
        toString: ƒ toString(),
        valueOf: ƒ valueOf()
    }
}
```

We can now use the `new` operator to create an instance of `doSomething()` based on this prototype. To use the new operator, simply call the function normally except prefix it with `new`. Calling a function with the `new` operator returns an object that is an instance of the function. Properties can then be added onto this object.

Try the following code:

```
function doSomething(){}
doSomething.prototype.foo = "bar";
var doSomeInstancing = new doSomething();
doSomeInstancing.prop = "some value";
console.log( doSomeInstancing );
```

This results in an output similar to the following:

```
{
    prop: "some value",
    __proto__: {
        foo: "bar",
        constructor: ƒ doSomething(),
        __proto__: {
            constructor: ƒ Object(),
            hasOwnProperty: ƒ hasOwnProperty(),
            isPrototypeOf: ƒ isPrototypeOf(),
            propertyIsEnumerable: ƒ propertyIsEnumerable(),
            toLocaleString: ƒ toLocaleString(),
            toString: ƒ toString(),
            valueOf: ƒ valueOf()
        }
    }
}
```

As seen above, the `__proto__` of `doSomeInstancing` is `doSomething.prototype`. But, what does this do? When you access a property of `doSomeInstancing`, the browser first looks to see if `doSomeInstancing` has that property.

If `doSomeInstancing` does not have the property, then the browser looks for the property in the `__proto__` of `doSomeInstancing` (a.k.a. doSomething.prototype). If the `__proto__` of doSomeInstancing has the property being looked for, then that property on the `__proto__` of doSomeInstancing is used.

Otherwise, if the `__proto__` of doSomeInstancing does not have the property, then the `__proto__` of the `__proto__` of doSomeInstancing is checked for the property. By default, the `__proto__` of any function's prototype property is `window.Object.prototype`. So, the `__proto__` of the `__proto__` of doSomeInstancing (a.k.a. the `__proto__` of doSomething.prototype (a.k.a. `Object.prototype`)) is then looked through for the property being searched for.

If the property is not found in the `__proto__` of the `__proto__` of doSomeInstancing, then the `__proto__` of the `__proto__` of the `__proto__` of doSomeInstancing is looked through. However, there is a problem: the `__proto__` of the `__proto__` of the `__proto__` of doSomeInstancing does not exist. Then, and only then, after the entire prototype chain of `__proto__`'s is looked through, and there are no more `__proto__`s does the browser assert that the property does not exist and conclude that the value at the property is `undefined`.

Let's try entering some more code into the console:

```
function doSomething(){}
doSomething.prototype.foo = "bar";
var doSomeInstancing = new doSomething();
doSomeInstancing.prop = "some value";
console.log("doSomeInstancing.prop:      " + doSomeInstancing.prop);
console.log("doSomeInstancing.foo:       " + doSomeInstancing.foo);
console.log("doSomething.prop:           " + doSomething.prop);
console.log("doSomething.foo:            " + doSomething.foo);
console.log("doSomething.prototype.prop: " + doSomething.prototype.prop);
console.log("doSomething.prototype.foo:  " + doSomething.prototype.foo);
```

This results in the following:

```
doSomeInstancing.prop:      some value
doSomeInstancing.foo:       bar
doSomething.prop:           undefined
doSomething.foo:            undefined
doSomething.prototype.prop: undefined
doSomething.prototype.foo:  bar
```

Different ways to create objects and the resulting prototype chain Section

Objects created with syntax constructs Section

```
var o = {a: 1};
```

```
var b = ['yo', 'whadup', '?'];
```

```
function f() {
  return 2;
}
```

With a constructor Section

A "constructor" in JavaScript is "just" a function that happens to be called with the new operator.

```
function Graph() {
  this.vertices = [];
  this.edges = [];
}

Graph.prototype = {
  addVertex: function(v) {
    this.vertices.push(v);
  }
};

var g = new Graph();
```

With `Object.create` Section

ECMAScript 5 introduced a new method: Object.create(). Calling this method creates a new object. The prototype of this object is the first argument of the function:

```
var a = {a: 1};


var b = Object.create(a);

console.log(b.a);

var c = Object.create(b);


var d = Object.create(null);

console.log(d.hasOwnProperty);
```

With the `class` keyword Section

ECMAScript 2015 introduced a new set of keywords implementing classes. The new keywords include class, constructor, static, extends, and super.

```
'use strict';

class Polygon {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}

class Square extends Polygon {
  constructor(sideLength) {
    super(sideLength, sideLength);
  }
  get area() {
    return this.height * this.width;
  }
  set sideLength(newLength) {
    this.height = newLength;
    this.width = newLength;
  }
}

var square = new Square(2);
```

Performance Section

The lookup time for properties that are high up on the prototype chain can have a negative impact on the performance, and this may be significant in the code where performance is critical. Additionally, trying to access nonexistent properties will always traverse the full prototype chain.

Also, when iterating over the properties of an object, **every** enumerable property that is on the prototype chain will be enumerated. To check whether an object has a property defined on *itself* and not somewhere on its prototype chain, it is necessary to use the hasOwnProperty method which all objects inherit from `Object.prototype`. To give you a concrete example, let's take the above graph example code to illustrate it:

```
console.log(g.hasOwnProperty('vertices'));

console.log(g.hasOwnProperty('nope'));

console.log(g.hasOwnProperty('addVertex'));

console.log(g.__proto__.hasOwnProperty('addVertex'));
```

hasOwnProperty is the only thing in JavaScript which deals with properties and does **not** traverse the prototype chain.

Note: It is **not** enough to check whether a property is undefined. The property might very well exist, but its value just happens to be set to `undefined`.

Bad practice: Extension of native prototypes 🔗 Section

One misfeature that is often used is to extend `Object.prototype` or one of the other built-in prototypes.

This technique is called monkey patching and breaks *encapsulation*. While used by popular frameworks such as Prototype.js, there is still no good reason for cluttering built-in types with additional *non-standard* functionality.

The **only** good reason for extending a built-in prototype is to backport the features of newer JavaScript engines, like `Array.forEach`.

Summary of methods for extending the prototype chain 🔗 Section

Here are all 4 ways and their pros/cons. All of the examples listed below create exactly the same resulting `inst` object (thus logging the same results to the console), except in different ways for the purpose of illustration.

Name	Example(s)	Pro(s)	Con(s)

New-initialization	`function foo(){}`
	`foo.prototype = {`
	`  foo_prop: "foo val"`
	`};`
	`function bar(){}`
	`var proto = new foo;`
	`proto.bar_prop = "bar val";`
	`bar.prototype = proto;`
	`var inst = new bar;`
	`console.log(inst.foo_prop);`
	`console.log(inst.bar_prop);`
		Supported in every browser imaginable (support goes all the way back to IE 5.5!). Also, it is very fast, very standard, and very JIST-optimizable.	In order to use this method, the function in question must be initialized. During this initialization, the constructor may store unique information that must be generated per-object. However, this unique information would only be generated once, potentially leading to problems. Additionally, the initialization of the constructor may put unwanted methods onto the object. However, both these are generally not problems at all (in fact, usually beneficial) if it is all your own code and you know what does what where.

Object.create	`function foo(){}`
	`foo.prototype = {`
	`  foo_prop: "foo val"`
	`};`
	`function bar(){}`
	`var proto = Object.create(`
	`  foo.prototype`
	`);`
	`proto.bar_prop = "bar val";`
	`bar.prototype = proto;`
	`var inst = new bar;`
	`console.log(inst.foo_prop);`
	`console.log(inst.bar_prop);`
	
	`function foo(){}`
	`foo.prototype = {`
	`  foo_prop: "foo val"`
	`};`
	`function bar(){}`
	`var proto = Object.create(`
	`  foo.prototype,`
	`  {`
	`    bar_prop: {`
	`      value: "bar val"`
	`    }`
	`  }`
	`);`
	`bar.prototype = proto;`
	`var inst = new bar;`
	`console.log(inst.foo_prop);`
	`console.log(inst.bar_prop)`
		Support in all in-use-today browsers which are all non-microsoft browsers plus IE9 and up. Allows the direct setting of __proto__ in a way that is one-time-only so that the browser can better optimize the object. Also allows the creation of objects without a prototype via `Object.create(null)` .	Not supported in IE8 and below. However, as Microsoft has discontinued extended support for systems running these old browsers, this should not be a concern for most applications. Additionally, the slow object initialization can be a performance black hole if using the second argument because each object-descriptor property has its own separate descriptor object. When dealing with hundreds of thousands of object descriptors in the form of object, there can arise a serious issue with lag.

Object.setPrototypeOf	`function foo(){} foo.prototype = { foo_prop: "foo val" }; function bar(){} var proto = { bar_prop: "bar val" }; Object.setPrototypeOf( proto, foo.prototype ); bar.prototype = proto; var inst = new bar; console.log(inst.foo_prop); console.log(inst.bar_prop);` `function foo(){} foo.prototype = { foo_prop: "foo val" }; function bar(){} var proto; proto=Object.setPrototypeOf( { bar_prop: "bar val" }, foo.prototype ); bar.prototype = proto; var inst = new bar; console.log(inst.foo_prop); console.log(inst.bar_prop)`	Support in all in-use-today browsers which are all non-microsoft browsers plus IE9 and up. Allows the dynamic manipulation of an objects prototype and can even force a prototype on a prototype-less object created with `Object.create(null)`.	Should-be-deprecated and ill-performant. Making your Javascript run fast is completely out of the question if you dare use this in the final production code because many browsers optimize the prototype and try to guess the location of the method in the memory when calling an instance in advance, but setting the prototype dynamically disrupts all these optimizations and can even force some browsers to recompile for deoptimization your code just to make it work according to the specs. Not supported in IE8 and below.
__proto__	`function foo(){} foo.prototype = { foo_prop: "foo val" }; function bar(){} var proto = { bar_prop: "bar val", __proto__: foo.prototype }; bar.prototype = proto; var inst = new bar; console.log(inst.foo_prop); console.log(inst.bar_prop);` `var inst = { __proto__: { bar_prop: "bar val", __proto__: { foo_prop: "foo val", __proto__: Object.prototype } } }; console.log(inst.foo_prop); console.log(inst.bar_prop)`	Support in all in-use-today browsers which are all non-microsoft browsers plus IE11 and up. Setting __proto__ to something that is not an object only fails silently. It does not throw an exception.	Grossly deprecated and non-performant. Making your Javascript run fast is completely out of the question if you dare use this in the final production code because many browsers optimize the prototype and try to guess the location of the method in the memory when calling an instance in advance, but setting the prototype dynamically disrupts all these optimizations and can even force some browsers to recompile for deoptimization your code just to make it work according to the specs. Not supported in IE10 and below.

`prototype` and `Object.getPrototypeOf` Section

JavaScript is a bit confusing for developers coming from Java or C++, as it's all dynamic, all runtime, and it has no classes at all. It's all just instances (objects). Even the "classes" we simulate are just a function object.

You probably already noticed that our `function A` has a special property called `prototype`. This special property works with the JavaScript `new` operator. The reference to the prototype object is copied to the internal `[[Prototype]]` property of the new instance. For example, when you do `var a1 = new A()`, JavaScript (after creating the object in memory and before running function `A()` with `this` defined to it) sets `a1.[[Prototype]] = A.prototype`. When you then access properties of the instance, JavaScript first checks whether they exist on that object directly, and if not, it looks in `[[Prototype]]`. This means that all the stuff you define in `prototype` is effectively shared by all instances, and you can even later change parts of `prototype` and have the changes appear in all existing instances, if you wanted to.

If, in the example above, you do `var a1 = new A(); var a2 = new A();` then `a1.doSomething` would actually refer to `Object.getPrototypeOf(a1).doSomething`, which is the same as the `A.prototype.doSomething` you defined, i.e. `Object.getPrototypeOf(a1).doSomething == Object.getPrototypeOf(a2).doSomething == A.prototype.doSomething`.

In short, `prototype` is for types, while `Object.getPrototypeOf()` is the same for instances.

`[[Prototype]]` is looked at *recursively*, i.e. `a1.doSomething`, `Object.getPrototypeOf(a1).doSomething`, `Object.getPrototypeOf(Object.getPrototypeOf(a1)).doSomething` etc., until it's found or `Object.getPrototypeOf` returns null.

So, when you call

```
var o = new Foo();
```

JavaScript actually just does

```
var o = new Object();
o.[[Prototype]] = Foo.prototype;
Foo.call(o);
```

(or something like that) and when you later do

```
o.someProp;
```

it checks whether `o` has a property `someProp`. If not, it checks `Object.getPrototypeOf(o).someProp`, and if that doesn't exist it checks `Object.getPrototypeOf(Object.getPrototypeOf(o)).someProp`, and so on.

In conclusion 🔗 Section

It is **essential** to understand the prototypal inheritance model before writing complex code that makes use of it. Also, be aware of the length of the prototype chains in your code and break them up if necessary to avoid possible performance problems. Further, the native prototypes should **never** be extended unless it is for the sake of compatibility with newer JavaScript features.