[[Prototype]] vs __proto__ vs .prototype in Javascript

Why are there so many types of prototypes in JavaScript and what do they all do?

4 min read · Jul 30, 2021

Eamon O'Callaghan Follow

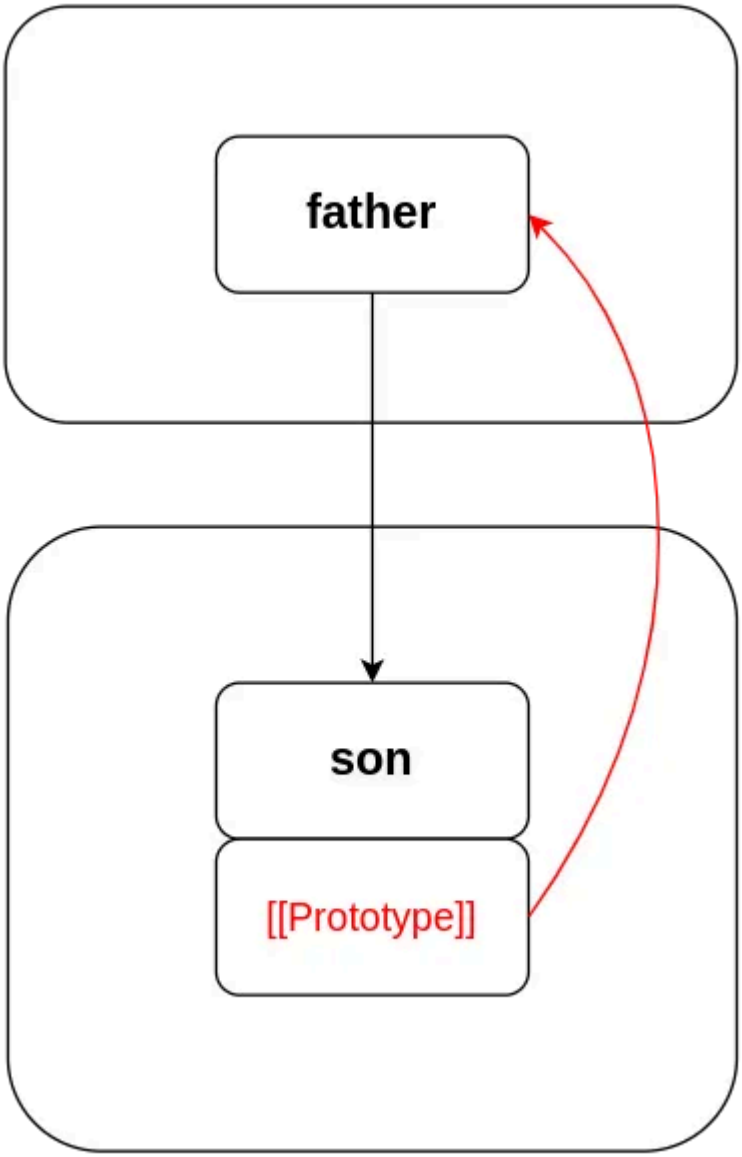
It can seem very daunting when you start learning about prototypes in JavaScript. A lot of the confusion stems from the fact that there are two different **prototypes** in JavaScript that refer to different concepts. Let me explain.

. . .

[[Prototype]]

`[[Prototype]]` is a hidden private property that all objects have in Javascript, it holds a reference to the object's **prototype**.

An object's prototype is the object that an object inherits or descends from. In the following diagram the object `son` descends from `father`, so `father` is the prototype of `son`. That means the hidden `[[Prototype]]` property of `son` points to `father`.



To summarize:

- `[[Prototype]]` is a hidden property that references an object's prototype.
- An object's prototype is the object that an object descends or inherits from.

. . .

__proto__

`__proto__` (also called the Dunder Proto or Double Underscore Prototype) is a property of `Object.prototype` (more on that in a minute) that exposes the hidden `[[Prototype]]` property of an object and allows you to access or modify it. You should not use it as it is **deprecated**, although you may come across it in older code.

The modern way of accessing an object's prototype is by using `Object.getPrototypeOf(obj)`. You can also modify an object's prototype using `Object.setPrototypeOf(obj, prototype)` as you can see in the following example:

```
let father = {};
let son = {};

Object.setPrototypeOf(son, father);
Object.getPrototypeOf(son); // father
```

To summarize:

- `__proto__` is a method that exposes the hidden `[[Prototype]]` property and allows you to modify it.
- `Object.getPrototypeOf()` and `Object.setPrototypeOf()` are the modern ways of getting access to and setting an object's prototype.

. . .

.prototype

`.prototype` is a special property that almost all functions have that is only used when a function is invoked as a constructor function. I say almost all because arrow functions and methods defined using the concise syntax do not have `.prototype` properties and cannot be used as constructors.

The `.prototype` property contains a reference to an object and when a constructor is used to instantiate or create a new object, `.prototype` is set as the prototype of the new object.

> *A constructor function is like an object factory that creates new objects which are instances of itself.* "What is a constructor function?"

For example, when the constructor `ObjectFactory` is used to instantiate a new object referenced by the variable `obj` on line 5 of the following code block, `obj`'s hidden internal `[[Prototype]]` property now holds a reference to the same object that is referenced by `ObjectFactory.prototype`. This means that any properties or methods defined in `ObjectFactory.prototype` are accessible by `obj.`

```
1 function ObjectFactory() {
2   this.property = `Hi, I'm a property!`;
3 }
4
5 let obj = new ObjectFactory();
6
7 console.log(typeof ObjectFactory.prototype); // object
8 console.log(ObjectFactory.prototype.isPrototypeOf(obj)); // true
```

Here on line 7, we prove that `ObjectFactory.prototype` references an object. On line 8 we prove that `ObjectFactory.prototype` is the prototype of `obj`.

Accessing a prototype's properties

```
9 // this is a continuation of the previous code block
10 ObjectFactory.prototype.prop = `I'm a property of ObjectFactory.prototype`;
11
12 console.log(obj); // ObjectFactory { property: "Hi, I'm a property!" }
13 console.log(obj.prop); // I'm a property of ObjectFactory.prototype
```

Notice that on line 10 we add a property called `prop` to the object referenced by `ObjectFactory.prototype` . Remember that `obj` ’s `[[Prototype]]` property references the same object as `ObjectFactory.prototype.`

We can see on line 12 that `obj` does not have a property called `prop`, it only has one property called `property`. However, `obj` can access `prop` on line 13 because it inherits the properties of its prototype which we know is the object referenced by `ObjectFactory.prototype`.

To summarize:

- `.prototype` is a special property that all functions have that contains a reference to an object.
- When a constructor is used to instantiate a new object, `ConstructorName.prototype` is set as the prototype of the new object.
- All instances of that constructor (the objects it creates) can access the properties of `ConstructorName.prototype`.

. . .

I briefly mentioned `Object.prototype` earlier in the article, `Object` is the built-in constructor function that almost all objects in JavaScript are instances of, and therefore almost all objects inherit from the object referenced by its `.prototype` property, `Object.prototype`.

. . .

I hope that this article has helped you to see the difference between these concepts! You can learn more about prototypical inheritance in JavaScript in my other article here.

Programming Prototype JavaScript Object Oriented Web Development