4th February 2020

Prototypal inheritance

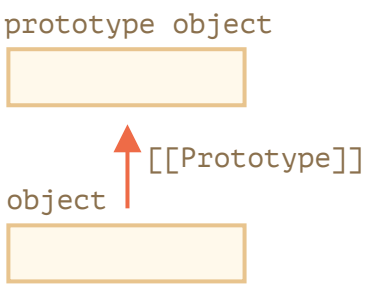
In programming, we often want to take something and extend it.

For instance, we have a `user` object with its properties and methods, and want to make `admin` and `guest` as slightly modified variants of it. We'd like to reuse what we have in `user`, not copy/reimplement its methods, just build a new object on top of it.

Prototypal inheritance is a language feature that helps in that.

[[Prototype]]

In JavaScript, objects have a special hidden property `[[Prototype]]` (as named in the specification), that is either `null` or references another object. That object is called "a prototype":



The prototype is a little bit "magical". When we want to read a property from `object`, and it's missing, JavaScript automatically takes it from the prototype. In programming, such thing is called "prototypal inheritance". Many cool language features and programming techniques are based on it.

The property `[[Prototype]]` is internal and hidden, but there are many ways to set it.

One of them is to use the special name `__proto__`, like this:

```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};

rabbit.__proto__ = animal;
```

`__proto__` is a historical getter/setter for `[[Prototype]]`

Please note that `__proto__` is *not the same* as `[[Prototype]]`. That's a getter/setter for it.

It exists for historical reasons. In modern language it is replaced with functions `Object.getPrototypeOf/Object.setPrototypeOf` that also get/set the prototype. We'll study the reasons for that and these functions later.

By the specification, `__proto__` must only be supported by browsers, but in fact all environments including server-side support it. For now, as `__proto__` notation is a little bit more intuitively obvious, we'll use it in the examples.

If we look for a property in `rabbit`, and it's missing, JavaScript automatically takes it from `animal`.

For instance:

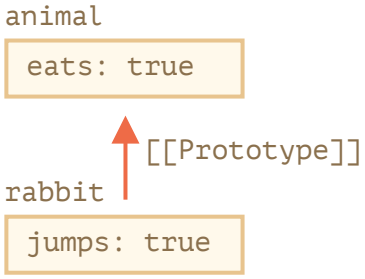
```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};

rabbit.__proto__ = animal; // (*)

// we can find both properties in rabbit now:
alert( rabbit.eats ); // true (**)
alert( rabbit.jumps ); // true
```

Here the line `(*)` sets `animal` to be a prototype of `rabbit`.

Then, when `alert` tries to read property `rabbit.eats` `(**)`, it's not in `rabbit`, so JavaScript follows the `[[Prototype]]` reference and finds it in `animal` (look from the bottom up):



Here we can say that "`animal` is the prototype of `rabbit`" or "`rabbit` prototypically inherits from `animal`".

So if `animal` has a lot of useful properties and methods, then they become automatically available in `rabbit`. Such properties are called "inherited".

If we have a method in `animal`, it can be called on `rabbit`:

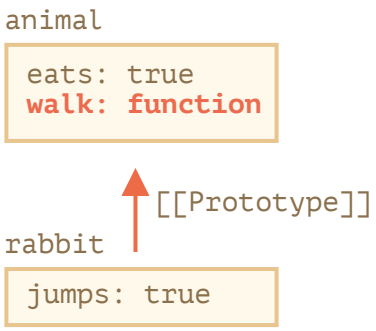
```
let animal = {
  eats: true,
  walk() {
```

```
    alert("Animal walk");
  }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

// walk is taken from the prototype
rabbit.walk(); // Animal walk
```

The method is automatically taken from the prototype, like this:



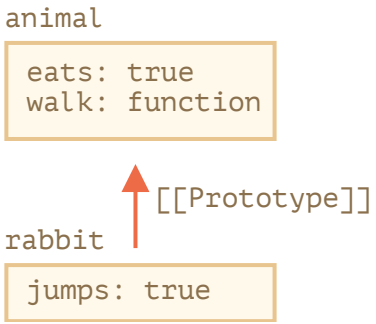
The prototype chain can be longer:

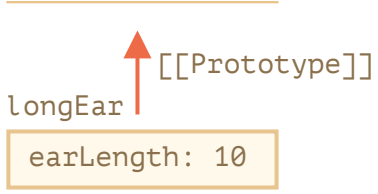
```
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

let longEar = {
  earLength: 10,
  __proto__: rabbit
};

// walk is taken from the prototype chain
longEar.walk(); // Animal walk
alert(longEar.jumps); // true (from rabbit)
```





There are only two limitations:

1. The references can't go in circles. JavaScript will throw an error if we try to assign `__proto__` in a circle.
2. The value of `__proto__` can be either an object or `null`. Other types are ignored.

Also it may be obvious, but still: there can be only one `[[Prototype]]`. An object may not inherit from two others.

Writing doesn't use prototype

The prototype is only used for reading properties.

Write/delete operations work directly with the object.

In the example below, we assign its own `walk` method to `rabbit`:

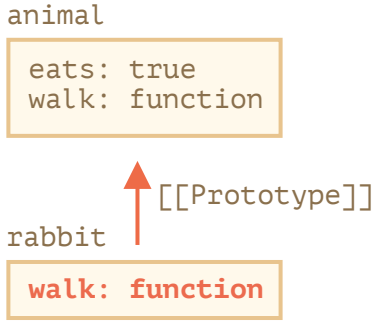
```
let animal = {
  eats: true,
  walk() {
    /* this method won't be used by rabbit */
  }
};

let rabbit = {
  __proto__: animal
};

rabbit.walk = function() {
  alert("Rabbit! Bounce-bounce!");
};

rabbit.walk(); // Rabbit! Bounce-bounce!
```

From now on, `rabbit.walk()` call finds the method immediately in the object and executes it, without using the prototype:



Accessor properties are an exception, as assignment is handled by a setter function. So writing to such a property is actually the same as calling a function.

For that reason `admin.fullName` works correctly in the code below:

```
let user = {
  name: "John",
  surname: "Smith",

  set fullName(value) {
    [this.name, this.surname] = value.split(" ");
  },

  get fullName() {
    return `${this.name} ${this.surname}`;
  }
};

let admin = {
  __proto__: user,
  isAdmin: true
};

alert(admin.fullName); // John Smith (*)

// setter triggers!
admin.fullName = "Alice Cooper"; // (**)
```

Here in the line `(*)` the property `admin.fullName` has a getter in the prototype `user`, so it is called. And in the line `(**)` the property has a setter in the prototype, so it is called.

The value of “this”

An interesting question may arise in the example above: what’s the value of `this` inside `set fullName(value)`? Where are the properties `this.name` and `this.surname` written: into `user` or `admin`?

The answer is simple: `this` is not affected by prototypes at all.

No matter where the method is found: in an object or its prototype. In a method call, `this` is always the object before the dot.

So, the setter call `admin.fullName=` uses `admin` as `this`, not `user`.

That is actually a super-important thing, because we may have a big object with many methods, and have objects that inherit from it. And when the inheriting objects run the inherited methods, they will modify only their own states, not the state of the big object.

For instance, here `animal` represents a “method storage”, and `rabbit` makes use of it.

The call `rabbit.sleep()` sets `this.isSleeping` on the `rabbit` object:

```
// animal has methods
let animal = {
  walk() {
    if (!this.isSleeping) {
      alert(`I walk`);
    }
  },
```

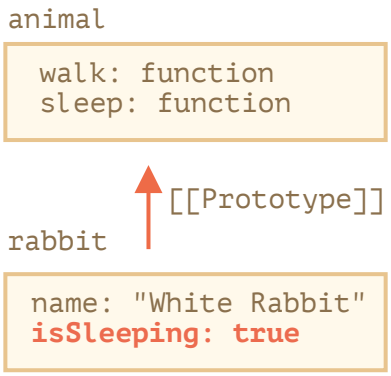
```
  sleep() {
    this.isSleeping = true;
  }
};

let rabbit = {
  name: "White Rabbit",
  __proto__: animal
};

// modifies rabbit.isSleeping
rabbit.sleep();

alert(rabbit.isSleeping); // true
alert(animal.isSleeping); // undefined (no such property in the prototype)
```

The resulting picture:



If we had other objects, like `bird`, `snake`, etc., inheriting from `animal`, they would also gain access to methods of `animal`. But `this` in each method call would be the corresponding object, evaluated at the call-time (before dot), not `animal`. So when we write data into `this`, it is stored into these objects.

As a result, methods are shared, but the object state is not.

for...in loop

The `for..in` loop iterates over inherited properties too.

For instance:

```
let animal = {
  eats: true
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

// Object.keys only returns own keys
alert(Object.keys(rabbit)); // jumps

// for..in loops over both own and inherited keys
for(let prop in rabbit) alert(prop); // jumps, then eats
```

If that's not what we want, and we'd like to exclude inherited properties, there's a built-in method obj.hasOwnProperty(key): it returns `true` if `obj` has its own (not inherited) property named `key`.

So we can filter out inherited properties (or do something else with them):

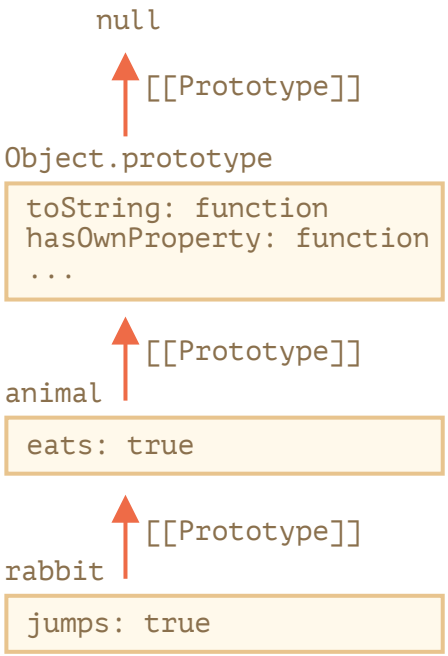
```
let animal = {
  eats: true
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

for(let prop in rabbit) {
  let isOwn = rabbit.hasOwnProperty(prop);

  if (isOwn) {
    alert(`Our: ${prop}`); // Our: jumps
  } else {
    alert(`Inherited: ${prop}`); // Inherited: eats
  }
}
```

Here we have the following inheritance chain: `rabbit` inherits from `animal`, that inherits from `Object.prototype` (because `animal` is a literal object `{...}`, so it's by default), and then `null` above it:



Note, there's one funny thing. Where is the method `rabbit.hasOwnProperty` coming from? We did not define it. Looking at the chain we can see that the method is provided by `Object.prototype.hasOwnProperty`. In other words, it's inherited.

...But why does `hasOwnProperty` not appear in the `for..in` loop like `eats` and `jumps` do, if `for..in` lists inherited properties?

The answer is simple: it's not enumerable. Just like all other properties of `Object.prototype`, it has `enumerable:false` flag. And `for..in` only lists enumerable properties. That's why it and the rest of the `Object.prototype` properties are not listed.

Almost all other key/value-getting methods ignore inherited properties

Almost all other key/value-getting methods, such as `Object.keys`, `Object.values` and so on ignore inherited properties.

They only operate on the object itself. Properties from the prototype are *not* taken into account.

Summary

- In JavaScript, all objects have a hidden `[[Prototype]]` property that's either another object or `null`.
- We can use `obj.__proto__` to access it (a historical getter/setter, there are other ways, to be covered soon).
- The object referenced by `[[Prototype]]` is called a "prototype".
- If we want to read a property of `obj` or call a method, and it doesn't exist, then JavaScript tries to find it in the prototype.
- Write/delete operations act directly on the object, they don't use the prototype (assuming it's a data property, not a setter).
- If we call `obj.method()`, and the `method` is taken from the prototype, `this` still references `obj`. So methods always work with the current object even if they are inherited.
- The `for..in` loop iterates over both its own and its inherited properties. All other key/value-getting methods only operate on the object itself.

Tasks

Working with prototype

importance: 5

Here's the code that creates a pair of objects, then modifies them.

Which values are shown in the process?

```
let animal = {
  jumps: null
};
let rabbit = {
  __proto__: animal,
  jumps: true
};

alert( rabbit.jumps ); // ? (1)

delete rabbit.jumps;

alert( rabbit.jumps ); // ? (2)

```

```
delete animal.jumps;

alert( rabbit.jumps ); // ? (3)
```

There should be 3 answers.

solution

Searching algorithm

importance: 5

The task has two parts.

Given the following objects:

```
let head = {
  glasses: 1
};

let table = {
  pen: 3
};

let bed = {
  sheet: 1,
  pillow: 2
};

let pockets = {
  money: 2000
};
```

1. Use `__proto__` to assign prototypes in a way that any property lookup will follow the path: `pockets` → `bed` → `table` → `head`. For instance, `pockets.pen` should be `3` (found in `table`), and `bed.glasses` should be `1` (found in `head`).
2. Answer the question: is it faster to get `glasses` as `pockets.glasses` or `head.glasses`? Benchmark if needed.

solution

Where does it write?

importance: 5

We have `rabbit` inheriting from `animal`.

If we call `rabbit.eat()`, which object receives the `full` property: `animal` or `rabbit`?

```
let animal = {
  eat() {
    this.full = true;
  }
};

let rabbit = {
  __proto__: animal
};

rabbit.eat();
```

solution

Why are both hamsters full?

importance: 5

We have two hamsters: `speedy` and `lazy` inheriting from the general `hamster` object.

When we feed one of them, the other one is also full. Why? How can we fix it?

```
let hamster = {
  stomach: [],

  eat(food) {
    this.stomach.push(food);
  }
};

let speedy = {
  __proto__: hamster
};

let lazy = {
  __proto__: hamster
};

// This one found the food
speedy.eat("apple");
alert( speedy.stomach ); // apple

// This one also has it, why? fix please.
alert( lazy.stomach ); // apple
```

solution

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)