21st October 2019

F.prototype

Remember, new objects can be created with a constructor function, like `new F()`.

If `F.prototype` is an object, then the `new` operator uses it to set `[[Prototype]]` for the new object.

> **Please note:**
>
> JavaScript had prototypal inheritance from the beginning. It was one of the core features of the language.
>
> But in the old times, there was no direct access to it. The only thing that worked reliably was a `"prototype"` property of the constructor function, described in this chapter. So there are many scripts that still use it.

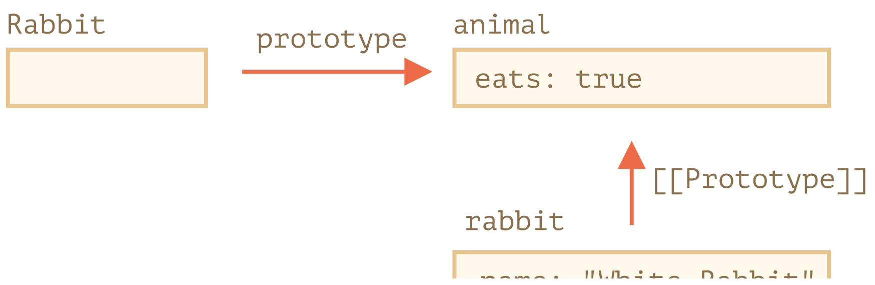
Please note that `F.prototype` here means a regular property named `"prototype"` on `F`. It sounds something similar to the term “prototype”, but here we really mean a regular property with this name.

Here’s the example:

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

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

Rabbit.prototype = animal;

let rabbit = new Rabbit("White Rabbit"); //  rabbit.__proto__ == animal

alert( rabbit.eats ); // true
```

Setting `Rabbit.prototype = animal` literally states the following: "When a `new Rabbit` is created, assign its `[[Prototype]]` to `animal`".

That’s the resulting picture:



On the picture, `"prototype"` is a horizontal arrow, meaning a regular property, and `[[Prototype]]` is vertical, meaning the inheritance of `rabbit` from `animal`.

`F.prototype` only used at `new F` time

`F.prototype` property is only used when `new F` is called, it assigns `[[Prototype]]` of the new object. After that, there's no connection between `F.prototype` and the new object. Think of it as a "one-time gift".

If, after the creation, `F.prototype` property changes (`F.prototype = <another object>`), then new objects created by `new F` will have another object as `[[Prototype]]`, but already existing objects keep the old one.

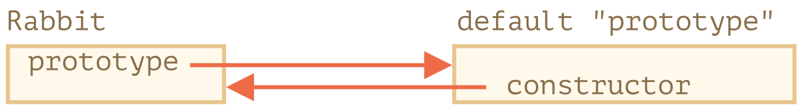
Default F.prototype, constructor property

Every function has the `"prototype"` property even if we don't supply it.

The default `"prototype"` is an object with the only property `constructor` that points back to the function itself.

Like this:

```
function Rabbit() {}

/* default prototype
Rabbit.prototype = { constructor: Rabbit };
*/
```



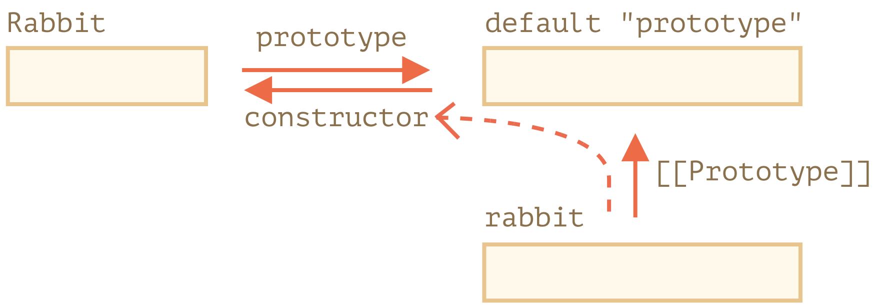
We can check it:

```
function Rabbit() {}
// by default:
// Rabbit.prototype = { constructor: Rabbit }

alert( Rabbit.prototype.constructor == Rabbit ); // true
```

Naturally, if we do nothing, the `constructor` property is available to all rabbits through `[[Prototype]]`:

```
function Rabbit() {}
// by default:
// Rabbit.prototype = { constructor: Rabbit }

let rabbit = new Rabbit(); // inherits from {constructor: Rabbit}

```

```
alert(rabbit.constructor == Rabbit); // true (from prototype)
```



We can use `constructor` property to create a new object using the same constructor as the existing one.

Like here:

```
function Rabbit(name) {
  this.name = name;
  alert(name);
}

let rabbit = new Rabbit("White Rabbit");

let rabbit2 = new rabbit.constructor("Black Rabbit");
```

That's handy when we have an object, don't know which constructor was used for it (e.g. it comes from a 3rd party library), and we need to create another one of the same kind.

But probably the most important thing about `"constructor"` is that…

…JavaScript itself does not ensure the right `"constructor"` value.

Yes, it exists in the default `"prototype"` for functions, but that's all. What happens with it later – is totally on us.

In particular, if we replace the default prototype as a whole, then there will be no `"constructor"` in it.

For instance:

```
function Rabbit() {}
Rabbit.prototype = {
  jumps: true
};

let rabbit = new Rabbit();
alert(rabbit.constructor === Rabbit); // false
```

So, to keep the right `"constructor"` we can choose to add/remove properties to the default `"prototype"` instead of overwriting it as a whole:

```
function Rabbit() {}

// Not overwrite Rabbit.prototype totally
// just add to it
Rabbit.prototype.jumps = true
// the default Rabbit.prototype.constructor is preserved
```

Or, alternatively, recreate the `constructor` property manually:

```
Rabbit.prototype = {
  jumps: true,
  constructor: Rabbit
};

// now constructor is also correct, because we added it
```

Summary

In this chapter we briefly described the way of setting a `[[Prototype]]` for objects created via a constructor function. Later we'll see more advanced programming patterns that rely on it.

Everything is quite simple, just a few notes to make things clear:

- The `F.prototype` property (don't mistake it for `[[Prototype]]`) sets `[[Prototype]]` of new objects when `new F()` is called.
- The value of `F.prototype` should be either an object or `null`: other values won't work.
- The `"prototype"` property only has such a special effect when set on a constructor function, and invoked with `new`.

On regular objects the `prototype` is nothing special:

```
let user = {
  name: "John",
  prototype: "Bla-bla" // no magic at all
};
```

By default all functions have `F.prototype = { constructor: F }`, so we can get the constructor of an object by accessing its `"constructor"` property.

Tasks

Changing "prototype"

importance: 5

In the code below we create `new Rabbit`, and then try to modify its prototype.

In the start, we have this code:

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();

alert( rabbit.eats ); // true
```

1.

We added one more string (emphasized). What will `alert` show now?

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();

Rabbit.prototype = {};

alert( rabbit.eats ); // ?
```

2.

…And if the code is like this (replaced one line)?

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();

Rabbit.prototype.eats = false;

alert( rabbit.eats ); // ?
```

3.

And like this (replaced one line)?

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();

delete rabbit.eats;

alert( rabbit.eats ); // ?
```

4.

The last variant:

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();

delete Rabbit.prototype.eats;

alert( rabbit.eats ); // ?
```

solution

Create an object with the same constructor

importance: 5

Imagine, we have an arbitrary object `obj`, created by a constructor function – we don't know which one, but we'd like to create a new object using it.

Can we do it like that?

```
let obj2 = new obj.constructor();
```

Give an example of a constructor function for `obj` which lets such code work right. And an example that makes it work wrong.

solution

Previous lesson Next lesson

Share Tutorial map

Comments