



🏠 → [The JavaScript language](#) → [Prototypes, inheritance](#)

📅 December 12, 2021

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.

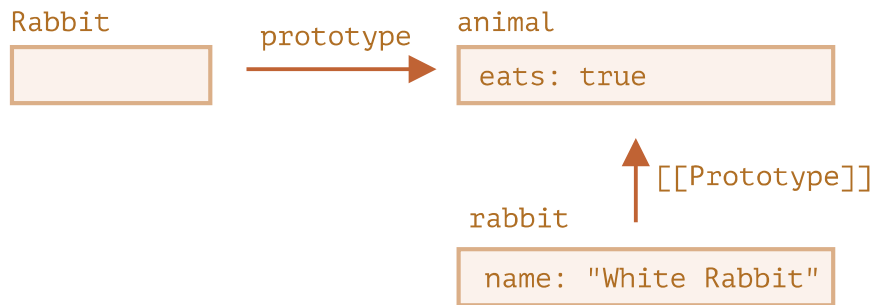
Please note that `F.prototype` here means a regular property named `"prototype"` on `F`. It sounds something similar to the term "prototype", but here we really mean a regular property with this name.

Here's the example:

```
1  let animal = {
2    eats: true
3  };
4
5  function Rabbit(name) {
6    this.name = name;
7  }
8
9  Rabbit.prototype = animal;
10
11 let rabbit = new Rabbit("White Rabbit"); // rabbit.__proto__ == animal
12
13 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.

i 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.

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:

```

1 function Rabbit() {}
2
3 /* default prototype
4 Rabbit.prototype = { constructor: Rabbit };
5 */
  
```



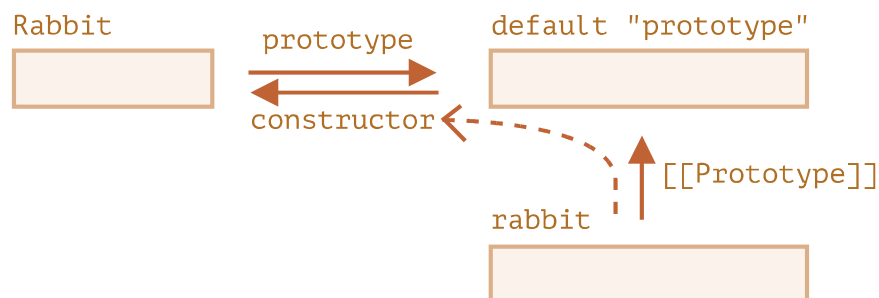
We can check it:

```

1 function Rabbit() {}
2 // by default:
3 // Rabbit.prototype = { constructor: Rabbit }
4
5 alert( Rabbit.prototype.constructor == Rabbit ); // true
  
```

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

```
1 function Rabbit() {}
2 // by default:
3 // Rabbit.prototype = { constructor: Rabbit }
4
5 let rabbit = new Rabbit(); // inherits from {constructor: Rabbit}
6
7 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:

```
1 function Rabbit(name) {
2   this.name = name;
3   alert(name);
4 }
5
6 let rabbit = new Rabbit("White Rabbit");
7
8 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:

```
1 function Rabbit() {}
2 Rabbit.prototype = {
3   jumps: true
4 };
5
6 let rabbit = new Rabbit();
7 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:

```
1 function Rabbit() {}
2
3 // Not overwrite Rabbit.prototype totally
4 // just add to it
5 Rabbit.prototype.jumps = true
6 // the default Rabbit.prototype.constructor is preserved
```

Or, alternatively, recreate the constructor property manually:

```
1 Rabbit.prototype = {
2   jumps: true,
3   constructor: Rabbit
4 };
5
6 // 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:

```
1 let user = {  
2   name: "John",  
3   prototype: "Bla-bla" // no magic at all  
4 };
```

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:

```
1 function Rabbit() {}  
2 Rabbit.prototype = {  
3   eats: true  
4 };  
5  
6 let rabbit = new Rabbit();  
7  
8 alert( rabbit.eats ); // true
```



1.

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

```
1 function Rabbit() {}  
2 Rabbit.prototype = {  
3   eats: true  
4 };  
5  
6 let rabbit = new Rabbit();  
7  
8 Rabbit.prototype = {};  
9  
10 alert( rabbit.eats ); // ?
```

2.

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

```
1 function Rabbit() {}
2 Rabbit.prototype = {
3   eats: true
4 };
5
6 let rabbit = new Rabbit();
7
8 Rabbit.prototype.eats = false;
9
10 alert( rabbit.eats ); // ?
```

3.

And like this (replaced one line)?

```
1 function Rabbit() {}
2 Rabbit.prototype = {
3   eats: true
4 };
5
6 let rabbit = new Rabbit();
7
8 delete rabbit.eats;
9
10 alert( rabbit.eats ); // ?
```

4.

The last variant:

```
1 function Rabbit() {}
2 Rabbit.prototype = {
3   eats: true
4 };
5
6 let rabbit = new Rabbit();
7
8 delete Rabbit.prototype.eats;
9
10 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?

```
1 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

- 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 few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox ([plnkr](#), [jsbin](#), [codepen](#)...)