10th December 2019

Class inheritance

Class inheritance is a way for one class to extend another class.

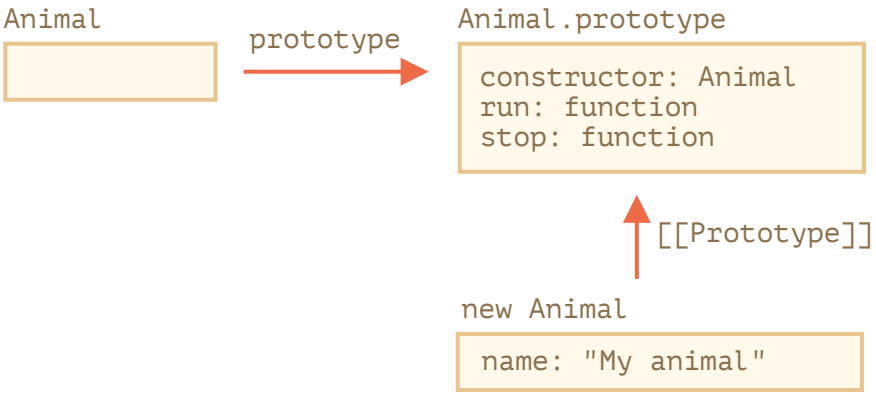
So we can create new functionality on top of the existing.

The “extends” keyword

Let’s say we have class `Animal`:

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed += speed;
    alert(`${this.name} runs with speed ${this.speed}.`);
  }
  stop() {
    this.speed = 0;
    alert(`${this.name} stands still.`);
  }
}

let animal = new Animal("My animal");
```

Here’s how we can represent `animal` object and `Animal` class graphically:



…And we would like to create another `class Rabbit`.

As rabbits are animals, `Rabbit` class should be based on `Animal`, have access to animal methods, so that rabbits can do what “generic” animals can do.

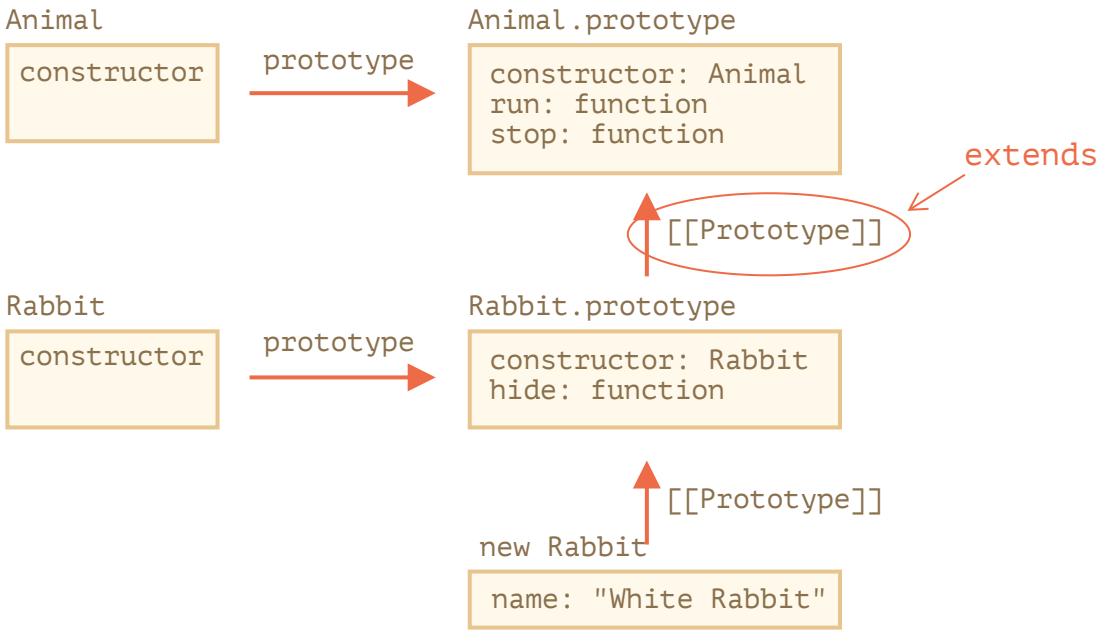
The syntax to extend another class is: `class Child extends Parent`.

Let's create `class Rabbit` that inherits from `Animal`:

```
class Rabbit extends Animal {
  hide() {
    alert(`${this.name} hides!`);
  }
}

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

rabbit.run(5); // White Rabbit runs with speed 5.
rabbit.hide(); // White Rabbit hides!
```

Object of `Rabbit` class have access to both `Rabbit` methods, such as `rabbit.hide()`, and also to `Animal` methods, such as `rabbit.run()`.

Internally, `extends` keyword works using the good old prototype mechanics. It sets `Rabbit.prototype.[[Prototype]]` to `Animal.prototype`. So, if a method is not found in `Rabbit.prototype`, JavaScript takes it from `Animal.prototype`.



For instance, to find `rabbit.run` method, the engine checks (bottom-up on the picture):

1. The `rabbit` object (has no `run`).
2. Its prototype, that is `Rabbit.prototype` (has `hide`, but not `run`).
3. Its prototype, that is (due to `extends`) `Animal.prototype`, that finally has the `run` method.

As we can recall from the chapter Native prototypes, JavaScript itself uses prototypal inheritance for built-in objects. E.g. `Date.prototype.[[Prototype]]` is `Object.prototype`. That's why dates have access to generic object methods.

Any expression is allowed after `extends`

Class syntax allows to specify not just a class, but any expression after `extends`.

For instance, a function call that generates the parent class:

```
function f(phrase) {
  return class {
    sayHi() { alert(phrase) }
  }
}

class User extends f("Hello") {}

new User().sayHi(); // Hello
```

Here `class User` inherits from the result of `f("Hello")`.

That may be useful for advanced programming patterns when we use functions to generate classes depending on many conditions and can inherit from them.

Overriding a method

Now let's move forward and override a method. By default, all methods that are not specified in `class Rabbit` are taken directly "as is" from `class Animal`.

But if we specify our own method in `Rabbit`, such as `stop()` then it will be used instead:

```
class Rabbit extends Animal {
  stop() {
    // ...now this will be used for rabbit.stop()
    // instead of stop() from class Animal
  }
}
```

Usually we don't want to totally replace a parent method, but rather to build on top of it to tweak or extend its functionality. We do something in our method, but call the parent method before/after it or in the process.

Classes provide `"super"` keyword for that.

- `super.method(...)` to call a parent method.
- `super(...)` to call a parent constructor (inside our constructor only).

For instance, let our rabbit autohide when stopped:

```
class Animal {

  constructor(name) {
    this.speed = 0;
    this.name = name;
  }

```

```
  run(speed) {
    this.speed += speed;
    alert(`${this.name} runs with speed ${this.speed}.`);
  }

  stop() {
    this.speed = 0;
    alert(`${this.name} stands still.`);
  }

}

class Rabbit extends Animal {
  hide() {
    alert(`${this.name} hides!`);
  }

  stop() {
    super.stop(); // call parent stop
    this.hide(); // and then hide
  }
}

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

rabbit.run(5); // White Rabbit runs with speed 5.
rabbit.stop(); // White Rabbit stands still. White rabbit hides!
```

Now `Rabbit` has the `stop` method that calls the parent `super.stop()` in the process.

Arrow functions have no `super`

As was mentioned in the chapter Arrow functions revisited, arrow functions do not have `super`.

If accessed, it's taken from the outer function. For instance:

```
class Rabbit extends Animal {
  stop() {
    setTimeout(() => super.stop(), 1000); // call parent stop after 1sec
  }
}
```

The `super` in the arrow function is the same as in `stop()`, so it works as intended. If we specified a "regular" function here, there would be an error:

```
// Unexpected super
setTimeout(function() { super.stop() }, 1000);
```

Overriding constructor

With constructors it gets a little bit tricky.

Until now, `Rabbit` did not have its own `constructor`.

According to the specification, if a class extends another class and has no `constructor`, then the following "empty" `constructor` is generated:

```
class Rabbit extends Animal {
  // generated for extending classes without own constructors
  constructor(...args) {
    super(...args);
  }
}
```

As we can see, it basically calls the parent `constructor` passing it all the arguments. That happens if we don't write a constructor of our own.

Now let's add a custom constructor to `Rabbit`. It will specify the `earLength` in addition to `name`:

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  // ...
}

class Rabbit extends Animal {

  constructor(name, earLength) {
    this.speed = 0;
    this.name = name;
    this.earLength = earLength;
  }

  // ...
}

// Doesn't work!
let rabbit = new Rabbit("White Rabbit", 10); // Error: this is not defined.
```

Whoops! We've got an error. Now we can't create rabbits. What went wrong?

The short answer is: constructors in inheriting classes must call `super(...)`, and (!) do it before using `this`.

…But why? What's going on here? Indeed, the requirement seems strange.

Of course, there's an explanation. Let's get into details, so you'll really understand what's going on.

In JavaScript, there's a distinction between a constructor function of an inheriting class (so-called "derived constructor") and other functions. A derived constructor has a special internal property `[[ConstructorKind]]:"derived"`. That's a special internal label.

That label affects its behavior with `new`.

- When a regular function is executed with `new`, it creates an empty object and assigns it to `this`.
- But when a derived constructor runs, it doesn't do this. It expects the parent constructor to do this job.

So a derived constructor must call `super` in order to execute its parent (non-derived) constructor, otherwise the object for `this` won't be created. And we'll get an error.

For the `Rabbit` constructor to work, it needs to call `super()` before using `this`, like here:

```
class Animal {

  constructor(name) {
    this.speed = 0;
    this.name = name;
  }

  // ...
}

class Rabbit extends Animal {

  constructor(name, earLength) {
    super(name);
    this.earLength = earLength;
  }

  // ...
}

// now fine
let rabbit = new Rabbit("White Rabbit", 10);
alert(rabbit.name); // White Rabbit
alert(rabbit.earLength); // 10
```

Super: internals, [[HomeObject]]

Advanced information

If you're reading the tutorial for the first time – this section may be skipped.

It's about the internal mechanisms behind inheritance and `super`.

Let's get a little deeper under the hood of `super`. We'll see some interesting things along the way.

First to say, from all that we've learned till now, it's impossible for `super` to work at all!

Yeah, indeed, let's ask ourselves, how it should technically work? When an object method runs, it gets the current object as `this`. If we call `super.method()` then, the engine needs to get the `method` from the prototype of the current object. But how?

The task may seem simple, but it isn't. The engine knows the current object `this`, so it could get the parent `method` as `this.__proto__.method`. Unfortunately, such a "naive" solution won't work.

Let's demonstrate the problem. Without classes, using plain objects for the sake of simplicity.

You may skip this part and go below to the `[[HomeObject]]` subsection if you don't want to know the details. That won't harm. Or read on if you're interested in understanding things in-depth.

In the example below, `rabbit.__proto__ = animal`. Now let's try: in `rabbit.eat()` we'll call `animal.eat()`, using `this.__proto__`:

```
let animal = {
  name: "Animal",
  eat() {
    alert(`${this.name} eats.`);
  }
};

let rabbit = {
  __proto__: animal,
  name: "Rabbit",
  eat() {
    // that's how super.eat() could presumably work
    this.__proto__.eat.call(this); // (*)
  }
};

rabbit.eat(); // Rabbit eats.
```

At the line `(*)` we take `eat` from the prototype (`animal`) and call it in the context of the current object. Please note that `.call(this)` is important here, because a simple `this.__proto__.eat()` would execute parent `eat` in the context of the prototype, not the current object.

And in the code above it actually works as intended: we have the correct `alert` .

Now let's add one more object to the chain. We'll see how things break:

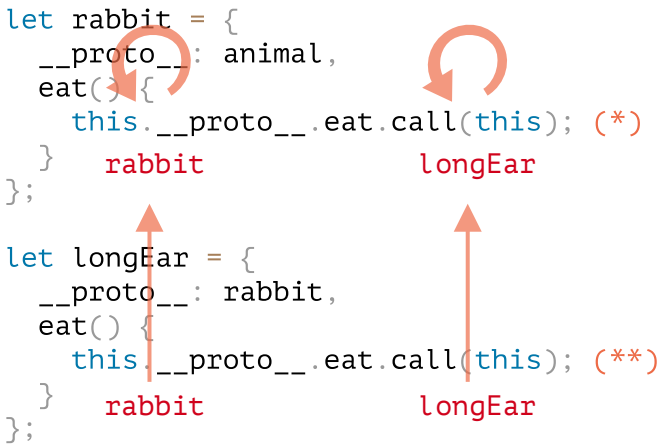
```
let animal = {
  name: "Animal",
  eat() {
    alert(`${this.name} eats.`);
  }
};

let rabbit = {
  __proto__: animal,
  eat() {
    // ...bounce around rabbit-style and call parent (animal) method
    this.__proto__.eat.call(this); // (*)
  }
};

let longEar = {
  __proto__: rabbit,
  eat() {
    // ...do something with long ears and call parent (rabbit) method
    this.__proto__.eat.call(this); // (**)
  }
};

longEar.eat(); // Error: Maximum call stack size exceeded
```

The code doesn't work anymore! We can see the error trying to call `longEar.eat()` .

It may be not that obvious, but if we trace `longEar.eat()` call, then we can see why. In both lines `(*)` and `(**)` the value of `this` is the current object (`longEar`). That's essential: all object methods get the current object as `this` , not a prototype or something.

So, in both lines `(*)` and `(**)` the value of `this.__proto__` is exactly the same: `rabbit`. They both call `rabbit.eat` without going up the chain in the endless loop.

Here's the picture of what happens:



1. Inside `longEar.eat()`, the line `(**)` calls `rabbit.eat` providing it with `this=longEar`.

```
// inside longEar.eat() we have this = longEar
this.__proto__.eat.call(this) // (**)
// becomes
longEar.__proto__.eat.call(this)
// that is
rabbit.eat.call(this);
```

2. Then in the line `(*)` of `rabbit.eat`, we'd like to pass the call even higher in the chain, but `this=longEar`, so `this.__proto__.eat` is again `rabbit.eat`!

```
// inside rabbit.eat() we also have this = longEar
this.__proto__.eat.call(this) // (*)
// becomes
longEar.__proto__.eat.call(this)
// or (again)
rabbit.eat.call(this);
```

3. …So `rabbit.eat` calls itself in the endless loop, because it can't ascend any further.

The problem can't be solved by using `this` alone.

[[HomeObject]]

To provide the solution, JavaScript adds one more special internal property for functions: `[[HomeObject]]`.

When a function is specified as a class or object method, its `[[HomeObject]]` property becomes that object.

Then `super` uses it to resolve the parent prototype and its methods.

Let's see how it works, first with plain objects:

```
let animal = {
  name: "Animal",
  eat() {         // animal.eat.[[HomeObject]] == animal
    alert(`${this.name} eats.`);
  }
};

let rabbit = {
  __proto__: animal,
  name: "Rabbit",
  eat() {         // rabbit.eat.[[HomeObject]] == rabbit
    super.eat();
  }
};

let longEar = {
  __proto__: rabbit,
  name: "Long Ear",
  eat() {         // longEar.eat.[[HomeObject]] == longEar
    super.eat();
  }
};

// works correctly
longEar.eat();  // Long Ear eats.
```

It works as intended, due to `[[HomeObject]]` mechanics. A method, such as `longEar.eat`, knows its `[[HomeObject]]` and takes the parent method from its prototype. Without any use of `this`.

Methods are not “free”

As we’ve known before, generally functions are “free”, not bound to objects in JavaScript. So they can be copied between objects and called with another `this`.

The very existance of `[[HomeObject]]` violates that principle, because methods remember their objects. `[[HomeObject]]` can’t be changed, so this bond is forever.

The only place in the language where `[[HomeObject]]` is used – is `super`. So, if a method does not use `super`, then we can still consider it free and copy between objects. But with `super` things may go wrong.

Here’s the demo of a wrong `super` result after copying:

```
let animal = {
  sayHi() {
    console.log(`I'm an animal`);
  }
};

// rabbit inherits from animal
let rabbit = {
  __proto__: animal,
  sayHi() {
    super.sayHi();
  }
};

```

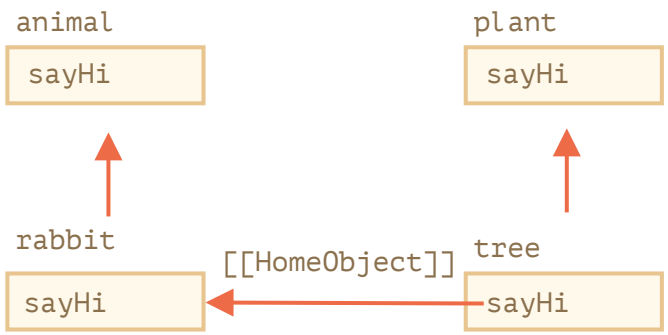
```
let plant = {
  sayHi() {
    console.log("I'm a plant");
  }
};

// tree inherits from plant
let tree = {
  __proto__: plant,
  sayHi: rabbit.sayHi // (*)
};

tree.sayHi();  // I'm an animal (?!?)
```

A call to `tree.sayHi()` shows "I'm an animal". Definitevely wrong.

The reason is simple:

- In the line `(*)`, the method `tree.sayHi` was copied from `rabbit`. Maybe we just wanted to avoid code duplication?
- Its `[[HomeObject]]` is `rabbit`, as it was created in `rabbit`. There's no way to change `[[HomeObject]]`.
- The code of `tree.sayHi()` has `super.sayHi()` inside. It goes up from `rabbit` and takes the method from `animal`.

Here's the diagram of what happens:



Methods, not function properties

`[[HomeObject]]` is defined for methods both in classes and in plain objects. But for objects, methods must be specified exactly as `method()`, not as `"method: function()"`.

The difference may be non-essential for us, but it's important for JavaScript.

In the example below a non-method syntax is used for comparison. `[[HomeObject]]` property is not set and the inheritance doesn't work:

```
let animal = {
  eat: function() { // intentionally writing like this instead of eat() {...
    // ...
  }
};

```

```
let rabbit = {
  __proto__: animal,
  eat: function() {
    super.eat();
  }
};

rabbit.eat();  // Error calling super (because there's no [[HomeObject]])
```

Summary

1. To extend a class: `class Child extends Parent`:
   - That means `Child.prototype.__proto__` will be `Parent.prototype`, so methods are inherited.
2. When overriding a constructor:
   - We must call parent constructor as `super()` in `Child` constructor before using `this`.
3. When overriding another method:
   - We can use `super.method()` in a `Child` method to call `Parent` method.
4. Internals:
   - Methods remember their class/object in the internal `[[HomeObject]]` property. That's how `super` resolves parent methods.
   - So it's not safe to copy a method with `super` from one object to another.

Also:

- Arrow functions don't have their own `this` or `super`, so they transparently fit into the surrounding context.

Tasks

Error creating an instance

importance: 5

Here's the code with `Rabbit` extending `Animal`.

Unfortunately, `Rabbit` objects can't be created. What's wrong? Fix it.

```
class Animal {

  constructor(name) {
    this.name = name;
  }

}

class Rabbit extends Animal {
  constructor(name) {
    this.name = name;
    this.created = Date.now();
  }
}
```

```
15
16 let rabbit = new Rabbit("White Rabbit"); // Error: this is not defined
17 alert(rabbit.name);
```

solution

Extended clock

importance: 5

We've got a `Clock` class. As of now, it prints the time every second.

```
1  class Clock {
2    constructor({ template }) {
3      this.template = template;
4    }
5
6    render() {
7      let date = new Date();
8
9      let hours = date.getHours();
10     if (hours < 10) hours = '0' + hours;
11
12     let mins = date.getMinutes();
13     if (mins < 10) mins = '0' + mins;
14
15     let secs = date.getSeconds();
16     if (secs < 10) secs = '0' + secs;
17
18     let output = this.template
19       .replace('h', hours)
20       .replace('m', mins)
21       .replace('s', secs);
22
23     console.log(output);
24   }
25
26   stop() {
27     clearInterval(this.timer);
28   }
29
30   start() {
31     this.render();
32     this.timer = setInterval(() => this.render(), 1000);
33   }
34 }
```

Create a new class `ExtendedClock` that inherits from `Clock` and adds the parameter `precision` – the number of `ms` between "ticks". Should be `1000` (1 second) by default.

- Your code should be in the file `extended-clock.js`
- Don't modify the original `clock.js`. Extend it.

Open a sandbox for the task.

solution

Class extends Object?

importance: 5

As we know, all objects normally inherit from `Object.prototype` and get access to "generic" object methods like `hasOwnProperty` etc.

For instance:

```
class Rabbit {
  constructor(name) {
    this.name = name;
  }
}

let rabbit = new Rabbit("Rab");

// hasOwnProperty method is from Object.prototype
alert( rabbit.hasOwnProperty('name') ); // true
```

But if we spell it out explicitly like `"class Rabbit extends Object"`, then the result would be different from a simple `"class Rabbit"`?

What's the difference?

Here's an example of such code (it doesn't work – why? fix it?):

```
class Rabbit extends Object {
  constructor(name) {
    this.name = name;
  }
}

let rabbit = new Rabbit("Rab");

alert( rabbit.hasOwnProperty('name') ); // true
```

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…)

© 2007—2020 Ilya Kantorabout the projectcontact usterms of usageprivacy policy