23rd November 2019

Static properties and methods

We can also assign a method to the class function itself, not to its `"prototype"`. Such methods are called *static*.

In a class, they are prepended by `static` keyword, like this:

```
class User {
  static staticMethod() {
    alert(this === User);
  }
}

User.staticMethod(); // true
```

That actually does the same as assigning it as a property directly:

```
class User { }

User.staticMethod = function() {
  alert(this === User);
};

User.staticMethod(); // true
```

The value of `this` in `User.staticMethod()` call is the class constructor `User` itself (the “object before dot” rule).

Usually, static methods are used to implement functions that belong to the class, but not to any particular object of it.

For instance, we have `Article` objects and need a function to compare them. A natural solution would be to add `Article.compare` method, like this:

```
class Article {
  constructor(title, date) {
    this.title = title;
    this.date = date;
  }

  static compare(articleA, articleB) {
    return articleA.date - articleB.date;
  }
}
```

```
11
12 // usage
13 let articles = [
14   new Article("HTML", new Date(2019, 1, 1)),
15   new Article("CSS", new Date(2019, 0, 1)),
16   new Article("JavaScript", new Date(2019, 11, 1))
17 ];
18
19 articles.sort(Article.compare);
20
21 alert( articles[0].title ); // CSS
```

Here `Article.compare` stands “above” articles, as a means to compare them. It’s not a method of an article, but rather of the whole class.

Another example would be a so-called “factory” method. Imagine, we need few ways to create an article:

1. Create by given parameters (`title` , `date` etc).
2. Create an empty article with today’s date.
3. …or else somehow.

The first way can be implemented by the constructor. And for the second one we can make a static method of the class.

Like `Article.createTodays()` here:

```
1  class Article {
2    constructor(title, date) {
3      this.title = title;
4      this.date = date;
5    }
6
7    static createTodays() {
8      // remember, this = Article
9      return new this("Today's digest", new Date());
10   }
11 }
12
13 let article = Article.createTodays();
14
15 alert( article.title ); // Today's digest
```

Now every time we need to create a today’s digest, we can call `Article.createTodays()` . Once again, that’s not a method of an article, but a method of the whole class.

Static methods are also used in database-related classes to search/save/remove entries from the database, like this:

```
1 // assuming Article is a special class for managing articles
2 // static method to remove the article:
3 Article.remove({id: 12345});
```

Static properties

A recent addition

This is a recent addition to the language. Examples work in the recent Chrome.

Static properties are also possible, they look like regular class properties, but prepended by `static`:

```
class Article {
  static publisher = "Ilya Kantor";
}

alert( Article.publisher ); // Ilya Kantor
```

That is the same as a direct assignment to `Article`:

```
Article.publisher = "Ilya Kantor";
```

Inheritance of static properties and methods

Static properties and methods are inherited.

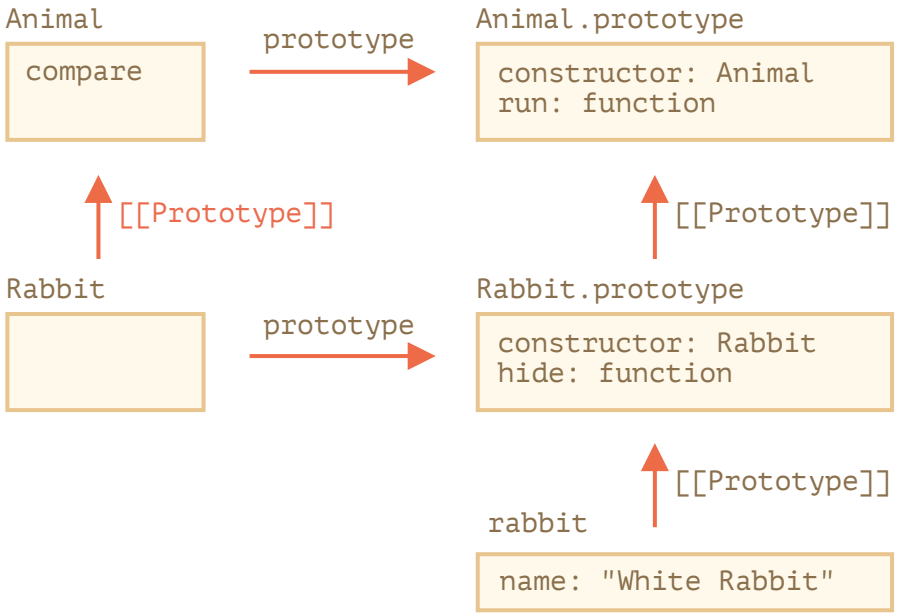
For instance, `Animal.compare` and `Animal.planet` in the code below are inherited and accessible as `Rabbit.compare` and `Rabbit.planet`:

```
class Animal {
  static planet = "Earth";

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

  run(speed = 0) {
    this.speed += speed;
    alert(`${this.name} runs with speed ${this.speed}.`);
  }

  static compare(animalA, animalB) {
    return animalA.speed - animalB.speed;
  }

}

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

let rabbits = [
  new Rabbit("White Rabbit", 10),
  new Rabbit("Black Rabbit", 5)
];
```

```
31
32 rabbits.sort(Rabbit.compare);
33
34 rabbits[0].run(); // Black Rabbit runs with speed 5.
35
36 alert(Rabbit.planet); // Earth
```

Now when we call `Rabbit.compare`, the inherited `Animal.compare` will be called.

How does it work? Again, using prototypes. As you might have already guessed, `extends` gives `Rabbit` the `[[Prototype]]` reference to `Animal`.



So, `Rabbit extends Animal` creates two `[[Prototype]]` references:

1. `Rabbit` function prototypally inherits from `Animal` function.
2. `Rabbit.prototype` prototypally inherits from `Animal.prototype`.

As a result, inheritance works both for regular and static methods.

Here, let's check that by code:

```
1 class Animal {}
2 class Rabbit extends Animal {}
3
4 // for statics
5 alert(Rabbit.__proto__ === Animal); // true
6
7 // for regular methods
8 alert(Rabbit.prototype.__proto__ === Animal.prototype); // true
```

Summary

Static methods are used for the functionality that belongs to the class "as a whole". It doesn't relate to a concrete class instance.

For example, a method for comparison `Article.compare(article1, article2)` or a factory method `Article.createTodays()` .

They are labeled by the word `static` in class declaration.

Static properties are used when we'd like to store class-level data, also not bound to an instance.

The syntax is:

```
class MyClass {
  static property = ...;

  static method() {
    ...
  }
}
```

Technically, static declaration is the same as assigning to the class itself:

```
MyClass.property = ...
MyClass.method = ...
```

Static properties and methods are inherited.

For `class B extends A` the prototype of the class `B` itself points to `A` : `B.[[Prototype]] = A` . So if a field is not found in `B` , the search continues in `A` .

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…)