JavaScript

this.fakeData = this.fakeData||{msg:"hello"}; //if fakeData is null, assign right side, otherwise self assign.

What you should know about `this`

`this` (aka "the context") is a special keyword inside each function and its value only depends on*how* the function was called, not how/when/where it was defined. It is not affected by lexical scope, like other variables. Here are some examples:

```
function foo(){
    console.log(this);}// normal function call
foo();// `this` will refer to `window`// as object methodvar obj ={bar: foo};
obj.bar();// `this` will refer to `obj`// as constructor functionnew foo();// `this` will refer to an object that inheri
ts from `foo.prototype`
```

To learn more about `this`, have a look at the MDN documentation.

How to refer to the correct `this`

Don't use `this`

You actually don't want to access `this` in particular, but *the object it refers to*. That's why an easy solution is to simply create a new variable that also refers to that object. The variable can have any name, but common ones are `self` and `that`.

```
functionMyConstructor(data, transport){this.data = data;varself=this;
    transport.on('data',function(){
        alert(self.data);});}
```

Since `self` is a normal variable, it obeys lexical scope rules and is accessible inside the callback. This also has the advantage that you can access the `this` value of the callback itself.

Explicitly set `this` of the callback - part 1

It might look like you have no control over the value of `this`, because its value is set automatically, but that is actually not the case.

Every function has the method `.bind` *[docs]*, which returns a new function with `this` bound to a value. The function has exactly the same behavior as the one you called `.bind` on, only that `this`was set by you. No matter how or when that function is called, `this` will always refer to the passed value.

```
functionMyConstructor(data, transport){this.data = data;var boundFunction =(function(){// parenthesis are not necessary
        alert(this.data);// but might improve readability}).bind(this);// <- here we are calling `.bind()`
    transport.on('data', boundFunction);}
```

In this case, we are binding the callback's `this` to the value of `MyConstructor`'s `this`.

Note: When binding context for jQuery, use `jQuery.proxy` *[docs]* instead. The reason to do this is so that you don't need to store the reference to the function when unbinding an event callback. jQuery handles that internally.

ECMAScript 6: Use arrow functions

ECMASCript 6 introduces *arrow functions*, which can be thought of as lambda functions. They don't have their own `this` binding. Instead, `this` is looked up in scope just like a normal variable. That means you don't have to call `.bind`. That's not the only special behavior they have, please refer to the MDN documentation for more information.

```
functionMyConstructor(data, transport){this.data = data;
    transport.on('data',()=> alert(this.data));}
```

Set `this` of the callback - part 2

Some functions/methods which accept callbacks also accept a value to which the callback's `this`should refer to. This is basically the same as binding it yourself, but the function/method does it for you. `Array#map` *[docs]* is such a method. Its signature is:

```
array.map(callback[, thisArg])
```

The first argument is the callback and the second argument is the value `this` should refer to. Here is a contrived example:

```
var arr =[1,2,3];var obj ={multiplier:42};var new_arr = arr.map(function(v){return v *this.multiplier;}, obj);// <- here
 we are passing `obj` as second argument
```

Note: Whether or not you can pass a value for `this` is usually mentioned in the documentation of that function/method. For example, jQuery's `$.ajax` method *[docs]* describes an option called `context`:

> This object will be made the context of all Ajax-related callbacks.

Common problem: Using object methods as callbacks / event handlers

Another common manifestation of this problem is when an object method is used as callback / event handler. Functions are first class citizens in JavaScript and the term "method" is just a colloquial term for a function that is a value of an object property. But that function doesn't have a specific link to its "containing" object.

Consider the following example:

```
functionFoo()[this.data =42,
    document.body.onclick =this.method;}Foo.prototype.method =function(){
    console.log(this.data);};
```

The function `this.method` is assigned as click event handler, but if the body is clicked, the value logged will be `undefined`, because inside the event handler, `this` refers to the body, not the instance of `Foo`.
As already mentioned at the beginning, what `this` refers to depends on how the function is **called**, not how it is **defined**.
If the code was like the following, it might be more obvious that the function doesn't have an implicit reference to the object:

```
function method(){
    console.log(this.data);}functionFoo()[this.data =42,
    document.body.onclick =this.method;}Foo.prototype.method = method;
```

The solution is the same as mentioned above: If available, use `.bind` to explicitly bind `this` to a specific value

```
document.body.onclick =this.method.bind(this);
```

or explicitly call the function as a "method" of the object, by using an anonymous function has callback / event handler and assign the object (`this`) to another variable:

```
varself=this;
document.body.onclick =function(){self.method();};
```

or use an arrow function:

```
document.body.onclick =()=>this.method();
```

Javascript语言不支持"类"，但是可以用一些变通的方法，模拟出"类"。

一、构造函数法

这是经典方法，也是教科书必教的方法。它用构造函数模拟"类"，在其内部用this关键字指代实例对象。

```
function Cat() {
    this.name = "大毛";
}
```

生成实例的时候，使用new关键字。

```
var cat1 = new Cat();
alert(cat1.name); // 大毛
```

类的属性和方法，还可以定义在构造函数的prototype对象之上。

```
Cat.prototype.makeSound = function(){
    alert("喵喵喵");
}
```

关于这种方法的详细介绍，请看我写的系列文章《Javascript 面向对象编程》，这里就不多说了。它的主要缺点是，比较复杂，用到了this和prototype，编写和阅读都很费力。

二、Object.create()法

为了解决"构造函数法"的缺点，更方便地生成对象，Javascript的国际标准ECMAScript第五版（目前通行的是第三版），提出了一个新的方法Object.create()。

用这个方法，"类"就是一个对象，不是函数。

```
var Cat = {
    name: "大毛",
    makeSound: function(){ alert("喵喵喵"); }
};
```

然后，直接用Object.create()生成实例，不需要用到new。

```
var cat1 = Object.create(Cat);
alert(cat1.name); // 大毛
cat1.makeSound(); // 喵喵喵
```

目前，各大浏览器的最新版本（包括IE9）都部署了这个方法。如果遇到老式浏览器，可以用下面的代码自行部署。

```
if (!Object.create) {
    Object.create = function (o) {
         function F() {}
        F.prototype = o;
        return new F();
    };
}
```

这种方法比"构造函数法"简单，但是不能实现私有属性和私有方法，实例对象之间也不能共享数据，对"类"的模拟不够全面。

三、极简主义法

荷兰程序员Gabor de Mooij提出了一种比Object.create()更好的新方法，他称这种方法为"极简主义法"（minimalist approach）。这也是我推荐的方法。

3.1 封装

这种方法不使用this和prototype，代码部署起来非常简单，这大概也是它被叫做"极简主义法"的原因。

首先，它也是用一个对象模拟"类"。在这个类里面，定义一个构造函数createNew()，用来生成实例。

```
var Cat = {
    createNew: function(){
        // some code here
    }
};
```

然后，在createNew()里面，定义一个实例对象，把这个实例对象作为返回值。

```
var Cat = {
    createNew: function(){
        var cat = {};
        cat.name = "大毛";
```

```
            cat.makeSound = function(){ alert("喵喵喵"); };
            return cat;
        }
    };
```

使用的时候，调用createNew()方法，就可以得到实例对象。

```
    var cat1 = Cat.createNew();
    cat1.makeSound(); // 喵喵喵
```

这种方法的好处是，容易理解，结构清晰优雅，符合传统的"面向对象编程"的构造，因此可以方便地部署下面的特性。

3.2 继承

让一个类继承另一个类，实现起来很方便。只要在前者的createNew()方法中，调用后者的createNew()方法即可。

先定义一个Animal类。

```
    var Animal = {
        createNew: function(){
            var animal = {};
            animal.sleep = function(){ alert("睡懒觉"); };
            return animal;
        }
    };
```

然后，在Cat的createNew()方法中，调用Animal的createNew()方法。

```
    var Cat = {
        createNew: function(){
    var cat = Animal.createNew();
            cat.name = "大毛";
            cat.makeSound = function(){ alert("喵喵喵"); };
            return cat;
        }
    };
```

这样得到的Cat实例，就会同时继承Cat类和Animal类。

```
    var cat1 = Cat.createNew();
    cat1.sleep(); // 睡懒觉
```

3.3 私有属性和私有方法

在createNew()方法中，只要不是定义在cat对象上的方法和属性，都是私有的。

```
    var Cat = {
        createNew: function(){
```

```
            var cat = {};
    var sound = "喵喵喵";
    cat.makeSound = function(){ alert(sound); };
            return cat;
        }
    };
```

上例的内部变量sound，外部无法读取，只有通过cat的公有方法makeSound()来读取。

```
    var cat1 = Cat.createNew();
alert(cat1.sound); // undefined
```

3.4 数据共享

有时候，我们需要所有实例对象，能够读写同一项内部数据。这个时候，只要把这个内部数据，封装在类对象的里面、createNew()方法的外面即可。

```
    var Cat = {
sound : "喵喵喵",
        createNew: function(){
            var cat = {};
cat.makeSound = function(){ alert(Cat.sound); };
cat.changeSound = function(x){ Cat.sound = x; };
            return cat;
        }
    };
```

然后，生成两个实例对象：

```
    var cat1 = Cat.createNew();
    var cat2 = Cat.createNew();
    cat1.makeSound(); // 喵喵喵
```

这时，如果有一个实例对象，修改了共享的数据，另一个实例对象也会受到影响。

```
cat2.changeSound("啦啦啦");
    cat1.makeSound(); // 啦啦啦
```

（完）

```
How to control namespace?
```

I use the approach found on the Enterprise jQuery site:

Here is their example showing how to declare private & public properties and functions. Everything is done as a self-executing anonymous function.

```
(function( skillet, $,undefined){//Private Propertyvar isHot =true;//Public Property
    skillet.ingredient ="Bacon Strips";//Public Method
    skillet.fry =function(){var oliveOil;

        addItem("\t\n Butter \n\t");
        addItem( oliveOil );
        console.log("Frying "+ skillet.ingredient );};//Private Methodfunction addItem( item ){if( item !==undefined){
```

```
            console.log("Adding "+ $.trim(item));}}}( window.skillet = window.skillet ||{}, jQuery ));
```

So if you want to access one of the public members you would just go `skillet.fry()` or `skillet.ingredients`.

What's really cool is that you can now extend the namespace using the exact same syntax.

```
//Adding new Functionality to the skillet(function( skillet, $,undefined){//Private Propertyvar amountOfGrease ="1 Cup";
//Public Method
    skillet.toString =function(){
        console.log( skillet.quantity +" "+
                     skillet.ingredient +" & "+
                     amountOfGrease +" of Grease");
        console.log( isHot ?"Hot":"Cold");};}( window.skillet = window.skillet ||{}, jQuery ));
```