**Наследование в JavaScript**

**Проблема**

Цель этой статьи - патологическая ненависть к велосипедам, добавляющим "классы" и "нормальное наследование" в JavaScript

Рассмотрим анти-пример (Class.create и Object.extend - функция из библиотеки Prototype.js)

// declare class

var ParentClass = Class.create();

Object.extend(ParentClass.prototype, {

    init : function() {},

    method : function() {}

});

// declare child class

var ChildClass = Class.create();

Object.extend(ChildClass.prototype, ParentClass.prototype):

Object.extend(ChildClass.prototype, {

    doSmthElse : function() {}

});

Чем этот код плох? А вот чем

(new ChildClass()) instanceof ParentClass === false;

То есть, по сути, здесь нет никакого наследования.

Если очень упрощенно, то наследование - это когда в потомке доступны public методы родителя.

А тут что происходит? Мы создали наследника и вручную скопировали в него все методы из родителя. Чем это плохо:

1. Если после такого "наследования" добавить метод в родительский класс - то в дочернем он будет не виден
2. В языках с динамической типизацией (например PHP) операция instanceof может быть использована для проверки типов параметров. Аналогично и в JavaScript. В частности можно объявить класс, у которого все методы бросают исключения - некое подобие интерфейса. А затем в методе проверять, что параметр instanceof этого класса. В случае же если операция instanceof не работает, то мы не сможем сделать "интерфейсы" и проверку типов.
3. на копирование методов нужно время. Если у нас N классов в каждом один новый метод, то общая сложность N\*N/2

Итак, рассмотрим, как правильно организовать наследование в JavaScript

**Giar 1**

/\* declare a class \*/

function ParentClass(name) {

    this.name = name;

}

// add method in the class

ParentClass.prototype.method = function() {

    alert(this.name);

}

// create new object of the class

var parent = new ParentClass('John');

// call method on the object

parent.method(); // alert('John');

/\* declare child class \*/

function ChildClass(name) {

    ParentClass.call(this, name); // call super

}

// inheritance trick

ChildClass.prototype = new ParentClass();

// add method in the class

ChildClass.prototype.newMethod = function() {

    this.method(); // can call methods inherited from parent

}

var child = new ChildClass('Smith');

child.method(); // can call methods inherited from parent

child instanceof ParentClass; // true

Все, этого достаточно. Это первый Gear, на нем можно ездить. Далее идет объяснение:

Наиболее близким понятием к "классу" в JavaScript'е является функция. Поэтому функция ParentClass (и ChildClass) одновременно является и объявлением нового класса и конструктором этого класса

В конструкторе класса ChildClass мы вызываем "супер" конструктор с помощью метода Function.call(context, param..). Если кратко, то это тоже самое что ParentClass(param), но только туда еще передается контекстный объект, то есть тот объект, на который будет указывать переменная this внутри вызываемой функции - ParentClass.call(this, param);

Наконец самое главное. Как так? Как это заработало? Что за трюк в строке

ChildClass.prototype = new ParentClass();

На самом деле все очень просто. Есть цепь прототипов. Когда мы пытаемся вызвать метод на объекте, то JavaScript в первую очередь проверят – есть ли такой метод на самом объекте? Если нет - то идет в прототип класса. Если там не находится нужный метод, тогда идем вверх по цепочке прототипов - либо пока не найдем, либо пока цепь не закончится. На вершине находится Object.prototype, а дальше null и цепь обрывается.

Как это работает? Опять же если кратко, то есть невидимая ссылка, ведущая вверх по цепочке прототипов. А начинается эта цепь с экземпляра класса:

parent.\_\_proto\_\_ === ParentClass.prototype

ParentClass.prototype.\_\_proto\_\_ === Object.prototype

Object.prototype.\_\_proto\_\_ === null

Таким образом, чтобы правильно организовать наследование нужно выстроить цепь прототипов:

function ParentClass(){}

function ChildClass(){}

ChildClass.prototype.\_\_proto\_\_ = ParentClass.prototype;

Но свойство \_\_proto\_\_ доступно только в WebKit браузерах, поэтому на него нельзя рассчитывать. Поэтому приходится делать трюк. Сравните:

// ChildClass.prototype.\_\_proto\_\_ = ParentClass.prototype;

ChildClass.prototype = new ParentClass();

**Gear 2**

Надоело ездить на первой скорости? Хотите private методы? Тогда включаем Gear 2.

function(){} // declare anonymous function

(function(){}) // around this function in braces, just for fun

(function(){})(); // call this function

// now lest try to write some code

// inside this anonymous function

(function(){

    // hey guys, this is a module!

})();

Итак, вот получился модуль, поместим внутрь него наш класс

(function() {

    var ParentClass = function(param) {

        this.param = param;

    }

    // ...

})();

Вот черт, теперь жеж этот объект не виден снаружи, не получится создать экземпляр класса. Давайте его опубликуем в глобальную область видимости (за глобальную область видимости отвечает переменная window)

window.ParentClass = ParentClass;

ну и все, теперь функции, объявленные внутри анонимной, но не опубликованные наружу будут private методами.

(function() {

    // pravate static field

    var uniqueNames = {};

    // private static method

    function \_checkNameUniqueness(name) {

        if (uniqueNames[name]) throw 'error';

        uniqueNames[name] = true;

    }

    // declare the class

    var ParentClass = function(name) {

        \_checkNameUniqueness(name);

        this.name = name;

    };

    window.ParentClass = ParentClass;

    // add method in the class

    ParentClass.prototype.method = function() {

        alert(this.name);

    }

})();

var parent = new ParentClass();

Только вот функция \_ckeckNameUniqueness() получилась "статическая" (в терминах Java). То есть в ней кючевое слово this имеет ссылку не на экземпляр класса ParentClass, а непонятко куда. Чтобы смотедлировать в JavaScript'е private non static method нужно обычный "статик" метод вызвать с помощью Function.call передав туда объект this

Все, можно забирать и пользоваться. А можно добавить еще всяких вкусностей.

1. Объявление класса и его публикацию можно сделать в одну строчку

window.ParentClass = function(name)

1. Добавить JS docs на методы и поля
2. При добавлении методв в прототип закалебешься писать каждый раз

ParentClass.prototype.method = ..,

ParentClass.prototype.method2 = ..,

можно воспользоваться какой-нибудь копирующей функцией типа Object.extend (из библиотеки prototype.js) или jQuery.extend (из библиотеки jQuery.js)

1. Ну и раз уж у нас JavaScript, то стало быть мы будем работать с DOM деревом. Для этого я выбираю библиотеку jQuery, но я так же знаю, что в NC широко используется prototype.js, и чтобы эти две библиотеки не конфликтовали приходится отказаться от алиаса '$' для jquery и везде писать полностью jQuery. А если все таки хочется использовать алиас? Можно передаться объект jQuery в нашу анонимную функцию, а уже внутри использовать алиас:

(function($){

    // use $

})(jQuery);

Итого получим

(function($) {

    /\*\*

     \* @type Object

     \*/

    var uniqueNames = {};

    /\*\*

     \* @static

     \* @param String name

     \*/

    function \_checkNameUniqueness(name) {

        if (uniqueNames[name]) throw 'error';

        uniqueNames[name] = true;

    }

    /\*\*

     \* @nonstatic

     \*/

    function \_getUpperName() {

        return this.name.toUpperCase();

    }

    /\*\*

     \* @constructor

     \* @param String name

     \*/

    window.ParentClass = function(name) {

        checkNameUniqueness(name);

        this.name = name;

    };

    // public methods

    jQuery.extend(ParentClass.prototype, {

        method : function() {

            // call non static private method

            var name = \_getUpperName.call(this);

            // alert(this.name);

            // now I can use jQuery alias !!!

            $('.dialog').html(this.name);

        },

    });

})(jQuery);

var parent = new ParentClass();

**Наследование в JavaScript (2)**

**Gear 3**

Черт, Лёха, ты нас обманул, твой трюк c наследованием не работает: из конструтора родительского класса валится эксепшн, поэтому я не могу сделать

ChildClass.prototype = new ParentClass();

И где обещанное удобство вызова супер методов?

Все есть, но только на третьей скорости. Включаем Gear 3.

function ParentClass() {throw 'error';}

function ChildClass() {}

// how to extend??

var child = new ChildClass();

child instaneof ParentClass; // DO SOMETHING, BUT IT MUST STAY 'true' !!!

Проблемы бы не было, если бы был доступно свойство \_\_proto\_\_. Но в общем случае его нет. new ParentClass() делать нельзя (кстати не только из-за эксепшна, а так же потому, что он может менять DOM дерево). Но при этом цепь прототипов должна быть. Что делать? Решение - добавить промежуточный класс, назовем его tmp

function tmp() {}

Отлично. Как отнаследовать tmp от ParentClass? Никак, но зато можно замаскироавть tmp так, чтобы он ничем не отличался от ParentClass

tmp.prototype = ParentClass.prototype;

Так, как теперь отнаследовать ChildClass от tmp? Класс tmp в своем конструкторе ничего плохого не делает, значит мы можем выстроить цепь наследования по старой схеме:

ChildClass.prototype = new tmp();

// putting it all togother:

function ParentClass() {throw 'error';}

function tmp() {}

tmp.prototype = ParentClass.prototype;

ChildClass.prototype = new tmp();

function ChildClass() {}

var child = new ChildClass();

child instanceof ParentClass;

Три строчки посредине, там где про tmp - это явный boilerplate код и его можно вынести в функцию inherit

function inherit(ChildClass, ParentClass) {

    function tmp() {};

    tmp.prototype = ParentClass.prototype;

    ChildClass.prototype = new tmp();

}

function ParentClass() {throw 'error';}

function ChildClass() {}

inherit(ChildClass, ParentClass);

var child = new ChildClass();

child instanceof ParentClass;

Фуф, наконец-то. Наследование работает, даже если в конструкторе родительского класса кидается эксепшн.

Рассмотрим еще один кейс. У нас есть объект obj и мы хотим создать объект такого же класса, как obj. Как это сделать? Можно воспользоваться свойством constructor. Это свойство есть абсолютно у всех объектов, т.к. оно приходит из Object.prototype

var constructor = obj.constructor;

var obj2 = new constructor();

Если попробовать выполнить предыдущий кусок кода, то можно заметить, что у объекта child "испорчена" ссылка на конструктор.

child.constructor === ParentClass; // a должно быть ChildClass

На самом деле эта бага тянется с самого первого гира. Но т.к. теперь у нас есть функция для наследования, то можно прямо в этой функции последней строкой поправить свойство constructor

ChildClass.prototype.constructor = ChildClass;

Ну и последнее. Ранее мы говорили о том, что из конструктора наследника можно вызвать конструктор родителя с помощью функции Function.call(context, param..)

function B(name){

    A.call(this, name);

}

все это хорошо и правильно, но что если мы изменим цепь наследования

A <- B

A <- X <- B

то есть теперь B не напрямую наследуется от A

И получится, что вызов A.call(this, name) надо переписывать! А ведь вызов родительского метода может быть не только из конструктора. Что же, все переписывать?

Нет. Давайте лучше где-нибудь сохраним ссылку на родительский класс. Пусть это будет свойство super. Удобнее всего его добавить в нашу функцию для наследования ChildClass.prototype.super = ParentClass;

function inherit(ChildClass, ParentClass) {

    function tmp() {};

    tmp.prototype = ParentClass.prototype;

    ChildClass.prototype = new tmp();

    ChildClass.prototype.constructor = ChildClass;

    ChildClass.prototype.super = ParentClass.prototype;

}

function ParentClass() {alert('test');}

function ChildClass() {this.super.constructor.call(this);}

inherit(ChildClass, ParentClass);

var child = new ChildClass();

Итого, для организации наследования в JavaScript'е не нужно использовать никаких сторонних библиотек типа prototype.js, потому что в них есть "правильное наследование", Class.create() и всякая чушь. Все что нужно это вот функция inherit из 7-ми строк. Ну и еще какая-нибудь функция extend, чтобы по десять раз не писать

B.prototype.func1 =..

B.prototype.func2 =..

**Gear 4**

Эй, Лёха, ты ведь все знаешь, а как сделать private instance поля? А то до сих пор у нас были только private static поля. Парни, включайте Gear 4

Тут уже конечно язык JavaScript себя исчерпал. Больше не осталось скрытых возможностей, которые бы можно было задействовать, чтобы "включить" private поля подобно тому, как мы включили private методы, нормальное наследование, проверку типов с помощью instanceof и свойство super для обращения к методам родителя.

Здесь мы будем использовать трюк с private static хэш таблицей.

Для начала создадим функцию generateId

(function(){

    var num = 0;

    window.generateId = function() {

        return num++;

    }

})();

Теперь напишем класс, который будет в конструкторе каждому объекту присваивать уникальный id. И заведем статическую мапу, где по id объекта будут храниться его поля. Получается некая "развертка" (тепрмин из мат.анализа)

(function(){

    var fields = {};

    window.MyClass = function(name) {

        this.id = generateId();

        fields[this.id] = {name:name};

    }

    MyClass.prototype.getName = function() {

        return fields[this.id].name;

    }

})();

Все, теперь у каждого объекта есть только одно public поле - это id. Ну и далее можно обратиться ко всем остальным полям с помощью fields[id].<private\_filed\_name> - это можно сделать только из методов класса. Снаружи - никак!

Поговорим о применимости всего этого. Когда все это нужно? Очевидно, что когда мы пишем фреймворк. Яркий пример CBTUI! В данный момент он используется на таком количестве проектов, что трудно уследить и очень трудно вносить изменения, нарушающие public API. (а ведь public поля - это public API). Еще пример - наши дашборды - со временем количество проектов, искользующих WidgetsFrm будет множиться. И постоянно пишутся свои кастомные виджеты, у которых есть своя кастомная JS часть. Еще пример - javascript компоненты обслуживающие parameters control и ingrid - от них тоже пишут проектные расширения.

Во всех этих случаях нужно грамотно продумывать архитектуру наследования и ограничивать видимость приватных членов классов. В будущем это позволит избежать лишних проблем и более просто рефакторить существующий код.