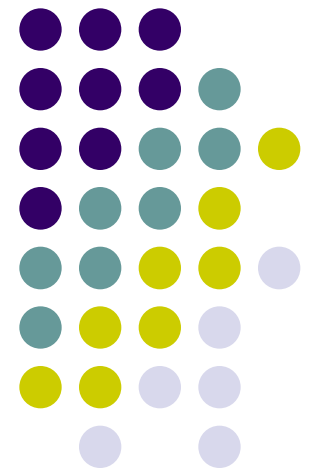


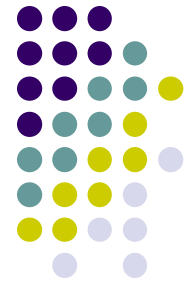
Javascript ООП

От и до





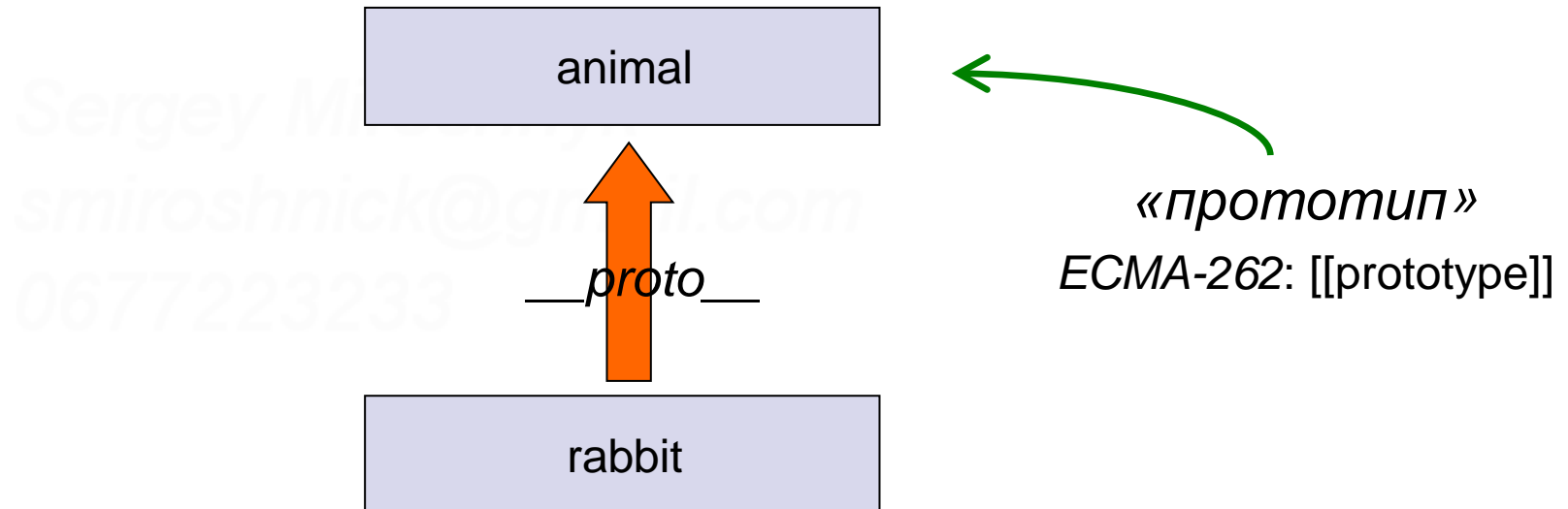
«КЛАССИЧЕСКОЕ» НАСЛЕДОВАНИЕ И ООП



Наследование

Объекты наследуют от объектов

- Один объект может иметь *неявную* ссылку на другой



- Свойства, которые не найдены в объекте, ищутся в `___proto___`



Пример `__proto__`

```
var animal = {  
  canWalk: true  
};
```

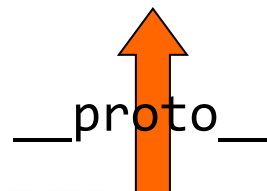
```
var rabbit = {  
  canJump: true  
};
```

```
rabbit.__proto__ = animal;
```

```
alert(rabbit.canWalk); // true
```

animal

canWalk: true



rabbit

canJump: true



Пример `__proto__`

```
var animal = {  
  canWalk: true  
};
```

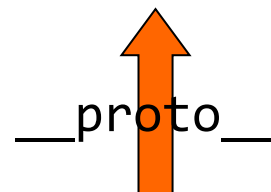
```
var rabbit = {  
  canJump: true  
};
```

```
rabbit.__proto__ = animal;
```

```
rabbit.canWalk = false;
```

animal

canWalk: true



`__proto__`

rabbit

canJump: true
canWalk: false

запись – напрямую
в объект

`__proto__` используется
только при чтении



Создание __proto__

Свойство prototype

```
animal = { canWalk: true };
```

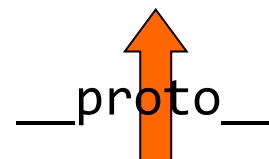
```
function Rabbit(name) {  
    this.name = name;  
}
```

```
Rabbit.prototype = animal;
```

```
rabbit = new Rabbit('Кроль');
```

animal

canWalk: true



rabbit


name: 'Кроль'

ссылка __proto__ создается
new по свойству prototype
конструктора Rabbit



Цепочка `__proto__`

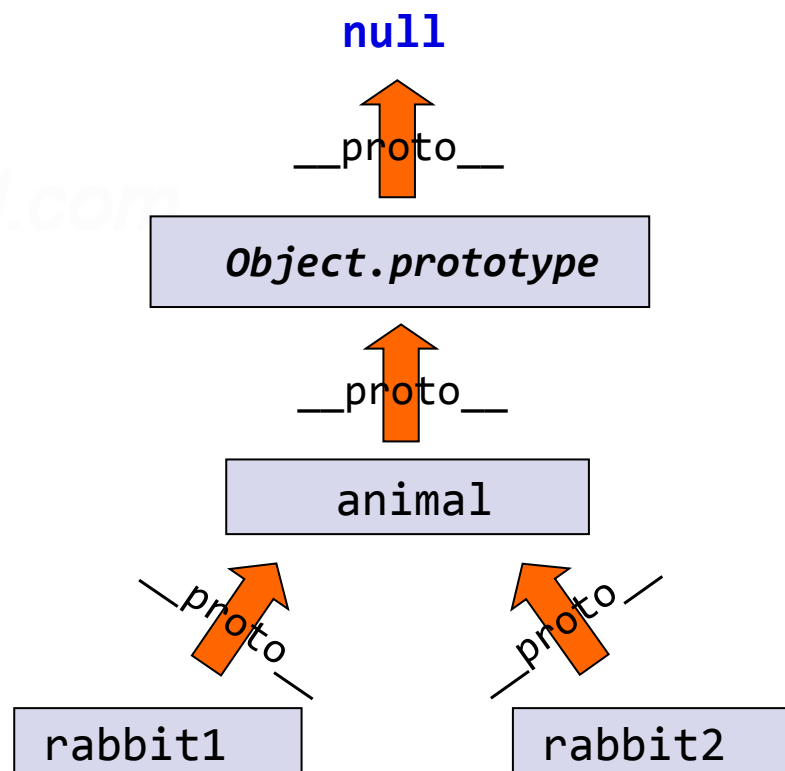
любой объект в итоге наследует `Object.prototype`

```
animal = { canWalk: true };  animal = new Object();  
animal.canWalk = true;
```

```
function Rabbit(name) {  
    this.name = name;  
}
```

```
Rabbit.prototype = animal;
```

```
rabbit1 = new Rabbit('Кроль');  
rabbit2 = new Rabbit('Заяц');
```



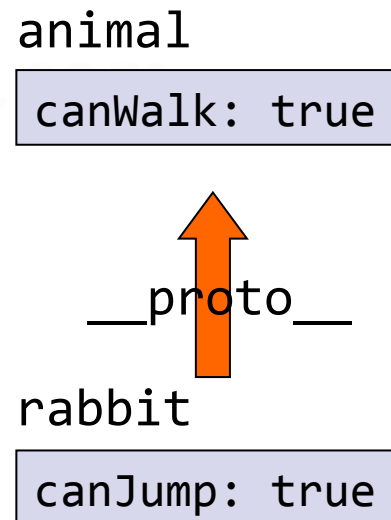


ES 5: создание объекта с прототипом

Object.create(proto[, props])

- Object.create(proto)
 - создает объект с заданным `__proto__`
 - работает в новых браузерах, IE9+, Opera 12

```
animal = {  
  canWalk: true;  
}  
  
rabbit = Object.create(animal);  
  
rabbit.canJump = true;  
  
alert(rabbit.canWalk); // true
```





ES3: inherit

Эмуляция Object.create

```
function inherit(proto) {  
  function F() { };  
  F.prototype = proto;  
  return new F;  
}
```

1. function F() { }



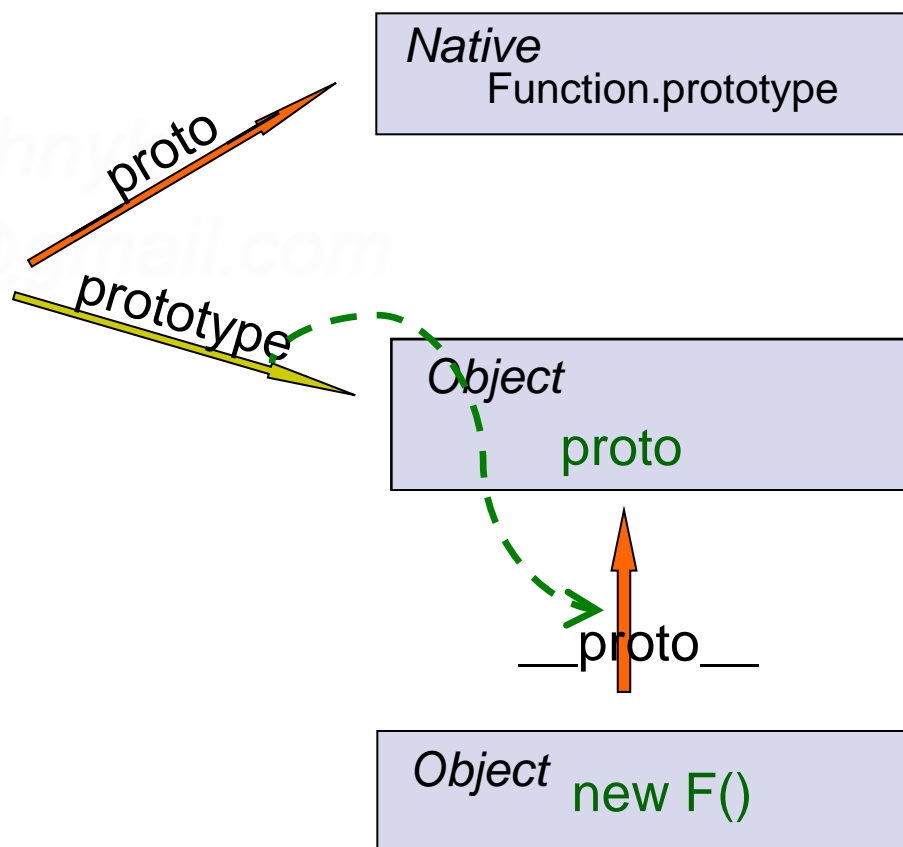
F = new Function()



2. F.prototype = proto



3. new F()





Наследование

псевдокласс вместо объекта

```
// --- класс Animal ---
function Animal(name) {
    this.name = name
}

// --- методы Animal ---
Animal.prototype.run = function() {
    alert("running")
}

// --- класс Rabbit ---
function Rabbit(name) {
    Animal.apply(this, arguments) // super()
}

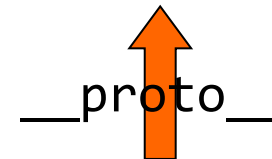
// наследование до объявления методов
Rabbit.prototype = inherit(Animal.prototype);

Rabbit.prototype.jump = function() {
    alert("jump!")
};

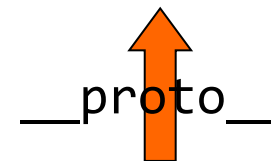
var rabbit = new Rabbit('Кроль')
```

Animal.prototype

run: function



Rabbit.prototype



rabbit

name: 'Кроль'

Методы хранятся в прототипе



inherit

Обзор

```
function inherit(proto) {  
    function F() { };  
    F.prototype = proto;  
    return new F;  
}
```

```
function Animal(name) {  
    this.name = name;  
}
```

Конструктор Animal

```
Animal.prototype.walk = function() {  
    alert(this + ": walking");  
};
```

Методы Animal

```
function Rabbit(name) {  
    Animal.apply(this, arguments);  
}
```

Конструктор Rabbit

```
Rabbit.prototype = inherit(Animal.prototype);
```

Унаследовали

```
Rabbit.prototype.jump = function() {  
    alert(this + ": jumping");  
}
```

Добавили методы Rabbit



extend

Альтернатива inherit

```
function extend(Child, Parent) {  
    function F() { }  
    F.prototype = Parent.prototype  
    Child.prototype = new F()  
}
```

```
function Animal(name) {  
    this.name = name;  
}
```

Конструктор Animal

```
Animal.prototype.walk = function() {  
    alert(this + ": walking");  
};
```

Методы Animal

```
function Rabbit(name) {  
    Animal.apply(this, arguments);  
}
```

Конструктор Rabbit

```
extend(Rabbit, Animal);
```

Унаследовали, extend делает:
Rabbit.prototype = inherit(Animal)

```
Rabbit.prototype.jump = function() {  
    alert(this + ": jumping");  
}
```

Методы Rabbit



Антипаттерн

```
function Animal(name) {  
    this.name = name;  
}
```

```
Animal.prototype.walk = function() {  
    alert(this + ": walking");  
};
```

```
function Rabbit(name) {  
    Animal.apply(this, arguments);  
}
```

```
Rabbit.prototype = new Animal();
```

```
Rabbit.prototype.jump = function() {  
    alert(this + ": jumping");  
}
```

Почему так –
плохо?



Проблема «недовиджета»

10 Родительский виджет

```
function Widget(id) {  
    var elem = document.getElementById(id)  
    /* ...создать виджет на основе элемента elem */  
}
```

10 Наследующий виджет

```
function Menu(id) { /* ... */ }
```

```
Menu.prototype = new Widget()
```

Ошибка при создании виджета,
т.к. `id=undefined`

И вообще,
зачем нам лишний
объект `Widget`?



instanceof использует __proto__

```
function Animal(name) {  
    this.name = name;  
}
```

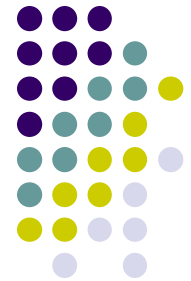
```
function Rabbit(name) {  
    Animal.apply(this, arguments);  
}
```

```
extend(Rabbit, Animal);
```

```
var rabbit = new Rabbit('Кроль');
```

```
alert(rabbit instanceof Rabbit); // true
```

```
alert(rabbit instanceof Animal); // true
```



constructor

В прототипе функции по умолчанию

Sergey Miroshnyk
smiroshnick@gmail.com
0677223233



Полиморфизм

перекрытие метода родителя

```
function Animal(name) {  
    this.name = name;  
}  
  
Animal.prototype.walk = function() {  
    alert("walking");  
};  
  
function Rabbit(name) {  
    Animal.apply(this, arguments);  
}  
  
extend(Rabbit, Animal);  
  
Rabbit.prototype.walk = function() {  
    Animal.prototype.walk.apply(this, arguments);  
    alert("... and jumping");  
}  
  
var rabbit = new Rabbit('Кроль');  
rabbit.walk(); // "walking", затем: "... and jumping"
```

Переопределили walk
С вызовом родительского walk



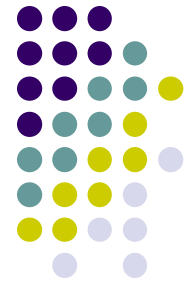
Инкапсуляция

защищенные свойства

соглашение об имени:

_prop
_method

```
function Animal(name) {  
    this.name = name  
}  
  
Animal.prototype._doWalk = function() { // protected  
    alert("running")  
}  
  
Animal.prototype.walk = function() { // public  
    this._doWalk()  
}
```



Синтаксический сахар

локализация имени родителя

parent.constructor...

```
function Rabbit(name) {  
  Animal.apply(this, arguments)  
}
```

имя родителя тут лишнее

parent.run...

```
// можно перекрыть метод родителя...  
Rabbit.prototype.walk = function() {  
  // вызвать родительский метод внутри  
  Animal.prototype.walk.apply(this, arguments)  
  alert("... and jumping")  
}
```



Синтаксический сахар

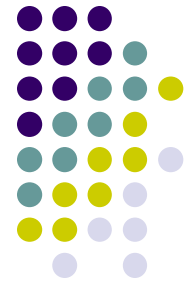
локализация имени родителя

```
function extend(Child, Parent) {  
    var F = function() { }  
    F.prototype = Parent.prototype  
    Child.prototype = new F()  
    // поправим свойство constructor  
    Child.prototype.constructor = Child  
    // добавим ссылку на методы родителя  
    Child.parent = Parent.prototype  
}  
  
function Rabbit(name) {  
    Rabbit.parent.constructor.apply(this, arguments) // конструктор родителя  
}  
  
extend(Rabbit, Animal)  
  
Rabbit.prototype.run = function() {  
    // вызвать родительский метод внутри  
    Rabbit.parent.run.apply(this, arguments)  
    alert("fast")  
}
```

Синтаксический сахар

`this._super`

`TODO: class.extend`



@see [Simple JavaScript Inheritance](#)

@see <http://ejohn.org/files/classy.js>

@see <http://learn.javascript.ru/files/tutorial/js/class-extend.js>



Статические свойства

10 Свойства объекта функции-конструктора

```
function Menu() {  
    Menu.menuCount = ++Menu.menuCount || 1  
}
```

```
var menu = new Menu()  
alert(Menu.menuCount)
```

```
var menu = new Menu()  
alert(Menu.menuCount)
```

Наследование



свойства-объекты



```
function Hamster() { }  
Hamster.prototype = {  
  food: [],  
  eat: function(something) {  
    this.food.push(something)  
  }  
}
```

// Создадим двух хомячков: speedy и lazy и накормим первого:

```
speedy = new Hamster()
```

```
lazy = new Hamster()
```

```
speedy.eat("apple")
```

```
speedy.eat("orange")
```

```
alert(speedy.food.length) // 2
```

```
alert(lazy.food.length) // 2 (!??)
```

- Пример



Наследование



свойства-объекты



```
function Hamster() {  
  this.food = []  
}  
Hamster.prototype = {  
  food: [], // @JSDOC  
  eat: function(something) {  
    this.food.push(something)  
  }  
}
```

```
speedy = new Hamster()  
lazy = new Hamster()
```

```
speedy.eat("apple")  
speedy.eat("orange")
```

```
alert(speedy.food.length) // 2  
alert(lazy.food.length) // 0(!)
```





БЕЗ ПРОТОТИПОВ

Sergey Smiroshnick
smiroshnick@gmail.com
0677223233



Наследование без прототипов

все объявления - в конструкторе

```
function Animal(name) {  
    this.name = name  
  
    this.run = function() {  
        alert("running")  
    }  
}  
  
function Rabbit(name) {  
    Animal.apply(this, arguments)  
    this.rabbitRun = function() {  
        alert("running fast!")  
    }  
}  
  
rabbit = new Rabbit("The Boss")
```

rabbit

name

run

rabbitRun

● [Демо](#)



Наследование без прототипов

приватные свойства и методы

```
function Animal(name) {  
    this.petName = name.toLowerCase();  
  
    var privateVar = 1;  
  
    function privateMethod() {  
        alert(this.petName);  
    }  
  
    this.run = function() {  
        alert(this.petName + " running");  
        privateMethod();  
    }  
}  
  
function Rabbit(name) {  
    // наследник не может пользоваться приватными методами  
    Animal.apply(this, arguments)  
}
```

← приватные

← неправильный this

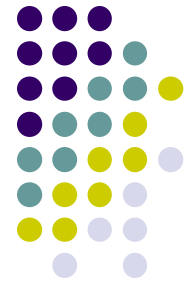


Наследование без прототипов

копирование объекта в self

```
function Animal(name) {  
    var self = this;  
  
    this.petName = name.toLowerCase();  
  
    var privateVar = 1;  
  
    function privateMethod() {  
        alert(self.petName);  
    }  
  
    this.run = function() {  
        alert(name + " running");  
        privateMethod();  
    };  
}
```

Для удобного рефакторинга лучше во всех методах использовать self



Наследование без прототипов

перекрытие

```
function Animal(name) {  
    this.run = function() {  
        alert(name + " running")  
        sayVar()  
    }  
}
```

```
function Rabbit(name) {  
    Animal.apply(this, arguments)
```

```
    var parentRun = this.run  
    this.run = function() {  
        parentRun() // чтобы работало - использовать self в родителе  
        // ...  
    }
```

```
    // перекрыть приватные методы нельзя
```

```
}
```



ФАБРИКА ОБЪЕКТОВ

Sergey Smiroshnick
smiroshnick@gmail.com
0677223233



Фабрика объектов (Python way)

```
var animal = Animal("Beast")  
animal.run()
```

~~new~~

```
function Animal(name) {  
  
    var privateVar = 5  
  
    function privateMethod() { /* ... */ }  
  
    return {  
  
        run: function() {  
            alert(name+ " running")  
        },  
        sit: function() {  
            return privateMethod()  
        }  
    }  
}
```

Как из приватного метода вызвать публичный?



Фабрика объектов

доступ к объекту из методов

```
function Animal(name) {  
    var privateVar = 5  
  
    function privateMethod() {  
        self.run() // приходится вводить переменную self  
    }  
    var self = {  
        run: function() {  
            alert(name+ " running")  
            this.sit() // внутри публичного метода доступен this  
        },  
        sit: function() {  
            // внутри privateMethod: this = window  
            privateMethod()  
        }  
    }  
    return self  
}
```




Фабрика объектов

наследование

```
function Animal(name) {  
  function privateMethod() { }  
  return {  
    run: function() {  
      alert(name + " running")  
    }  
  }  
}
```

```
function Rabbit(name) {  
  var rabbit = Animal(name)  
  rabbit.jump = function() {  
    alert("jump!")  
  }  
  var parentRun = rabbit.run  
  rabbit.run = function() {  
    parentRun.call(this)  
    alert("fast")  
  }  
  return rabbit  
}
```

унаследовать

добавить метод

перекрыть родителя

```
r1 = Rabbit("chuk")  
r2 = Rabbit("gek")
```

● Пример



Фабрика объектов VS *this*

```
function Animal(name) {  
    //...  
}
```

```
function Rabbit(name) {  
    var rabbit = Animal(name);  
  
    rabbit.jump = function() {  
        alert("jump!");  
    };  
  
    var parentRun = rabbit.run;  
    rabbit.run = function() {  
        parentRun();  
        alert("fast");  
    };  
  
    return rabbit;  
}
```

```
rabbit = Rabbit("chuk");  
- не работает instanceof
```

```
function Animal(name) {  
    //...  
}
```

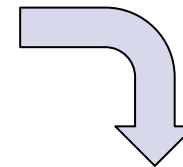
```
function Rabbit(name) {  
    Animal.apply(this, arguments);  
  
    this.jump = function() {  
        alert("jump!");  
    };  
  
    var parentRun = this.run;  
    this.run = function() {  
        parentRun();  
        alert("fast");  
    };  
}
```

```
rabbit = new Rabbit("chuk");
```

Классическое наследование VS Наследование без прототипов



- + приватные свойства и методы
 - + *удобно и полезно*
 - + *безопасное сжатие*
 - + *не возникает вопросов с хомьяками*
- не работает instanceof
- медленнее при создании,
- ест больше памяти, создает ссылки
- не отделяется конструктор





Конструктор + Методы = ☹

```
function Menu(id) {  
  render();  
  
  var render = this._render = function() {  
    $('#id').click(...)  
  };  
}
```

```
function SlidingMenu(id) {  
  Menu.apply(this, arguments);  
  
  // хочу переопределить _render  
  // ... но конструктор уже выполнен!  
}
```

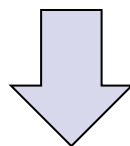
конструктор

методы родителя

конструктор родителя

методы родителя

методы потомка



- конструктор родителя не может использовать переопределенные методы
- конструктор родителя нельзя переопределить



Резюме

- ⑩ Если наследование не ожидается
 - => наследование «без прототипов»
 - бонус в виде приватных свойств и методов
 - хорошо сжимаются
- ⑩ Для полноценного наследования
 - => наследование на прототипах



СИНГЛТОН

```
var Singleton = new function() {  
  
    var privateVar;  
  
    function privateMethod() {  
  
    }  
  
    this.publicMethod = function() {  
  
    };  
  
}
```



Проверка типов instanceof

⑩ jQuery

```
isFunction: function( obj ) {  
    return {}.toString.call(obj) === "[object Function]";  
},
```

```
isArray: function( obj ) {  
    return {}.toString.call(obj) === "[object Array]";  
},
```

...

⑩ Почему не

- `arr instanceof Array?`



Проверка типов

`instanceof`

- ⑩ Все данные и типы привязаны к окну:
 - [Демо для Array](#)
 - то же самое для Date и остальных типов

Sergey Miroshnyk
smiroshnick@gmail.com
0677223233



Проверка типов `instanceof`

- ⑩ способ 1: проверка с указанием окна:

```
value instanceof otherWin.Array => true
```

- ⑩ способ 2: проверка типовых методов:

```
if (value.splice) => Array
```

```
if (value.toGMTString) => Date
```

- ⑩ способ 3: нативные объекты имеют `[[Class]]`

```
{}.toString.call(obj) дает [[Class]] в обертке
```

```
{}.toString.call(new Array) = [object Array]
```

Object.prototype.toString