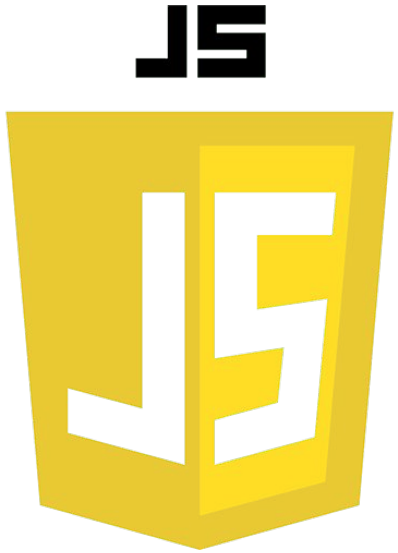
iTechArt

Software
Engineering
Services

October 17, 2017

JAVA SCRIPT
CRITICAL PARTS



ANTON CHERNOV

ЦЕЛИ ЗАНЯТИЯ

1. Более детальное знакомство с Javascript
2. Поднятие (hoisting)
3. Контекст выполнения
4. Замыкания
5. Модули
6. Наследование

HOISTING

ПОДНЯТИЕ (HOISTING)

Объявления переменных и функций попадают в память в процессе фазы компиляции, но остаются в коде на том месте, где вы их объявили

CODE SNIPPET

```
catName("Раиса");

function catName(name) {
  console.log("Мою кошку зовут " + name);
}
/*
 Результатом будет вывод строки: "Мою кошку зовут Раиса"
*/
```

ПОДНЯТИЕ (HOISTING)

Объявления переменных и функций попадают в память в процессе фазы компиляции, но остаются в коде на том месте, где вы их объявили

CODE SNIPPET

```
var x = 1; // Инициализируем x
console.log(x + " " + y);  // '1 undefined'
var y = 2;
//код выше и код ниже одинаковые

var x = 1; // Инициализируем x
var y; // Объявляем y
console.log(x + " " + y);  // '1 undefined'
y = 2; // Инициализируем y
```

ES5 STRICT MODE

Режим *strict* (*строгий* режим), введенный в ECMAScript 5, позволяет использовать более строгий вариант JavaScript.

CODE SNIPPET

```
// Синтаксис переключения в строгий режим всего скрипта
"use strict";
var v = "Привет! Я скрипт в строгом режиме!";

function strict() {
  // Strict режим на уровне функции
  'use strict';
  function nested() { return "And so am I!"; }
  return "Hi! I'm a strict mode function!  " + nested();
}
function notStrict() { return "I'm not strict."; }
```

КОНТЕКСТ ВЫПОЛНЕНИЯ

ГЛОБАЛЬНЫЙ КОНТЕКСТ

В глобальном контексте выполнения (за пределами каких-либо функций), this ссылается на глобальный объект вне зависимости от использования в строгом или нестрогом режиме.

CODE SNIPPET

```
console.log(this.document === document); // true

// В браузерах, объект window также является
глобальным:
console.log(this === window); // true

this.a = 37;
console.log(window.a); // 37
```

КОНТЕКСТ ФУНКЦИИ
ПРОСТОЙ ВЫЗОВ

В пределах функции значение this зависит от того, каким образом вызвана функция.

CODE SNIPPET

```
"use strict"; // see strict mode

function f2(){
  return this;
}

f2() === undefined; // true

window.f2() === window; // true
```

КОНТЕКСТ ФУНКЦИИ
МЕТОД ОБЪЕКТА

Когда функция вызывается как метод объекта, используемое в этой функции ключевое слово this принимает значение объекта, по отношению к которому вызнан метод.

CODE SNIPPET

```
var o = {
  prop: 37,
  f: function () {
    return this.prop;
  }
};

console.log(o.f()); // logs 37
```

КОНТЕКСТ ФУНКЦИИ
МЕТОД ОБЪЕКТА

Когда функция вызывается как метод объекта, используемое в этой функции ключевое слово this принимает значение объекта, по отношению к которому вызнан метод.

CODE SNIPPET

```
var o = {prop: 37};

function independent() {
  return this.prop;
}

o.f = independent;

console.log(o.f()); // logs 37
```

```
o.b = {
  g: independent,
  prop: 42
};

console.log(o.b.g());
// logs 42
```

КОНТЕКСТ ФУНКЦИИ
CALL AND APPLY

Когда в теле функции используется ключевое слово this , его значение может быть привязано к конкретному объекту в вызове при помощи методов call или apply, которые наследуются всеми функциями от Function.prototype.

CODE SNIPPET

```
function add(c, d){
  return this.a + this.b + c + d;
}

var o = {a:1, b:3};

add.call(o, 5, 7); // 1 + 3 + 5 + 7 = 16

add.apply(o, [10, 20]); // 1 + 3 + 10 + 20 = 34
```

КОНТЕКСТ ФУНКЦИИ
BIND

Вызов f.bind(someObject) создает новую функцию с тем же телом и областью видимости что и f, но там, где находится this в исходной функции, в новой функции существует постоянная привязка к первому аргументу метода bind, несмотря на то, как используется данная функция.

CODE SNIPPET

```
function f(){
  return this.a;
}

var g = f.bind({a:"azerty"});
console.log(g()); // azerty

var o = {a:37, f:f, g:g};
console.log(o.f(), o.g()); // 37, azerty
```

КОНТЕКСТ ФУНКЦИИ
КАРРИНГ **(CURRYING)**

CODE SNIPPET

```
function mul(a, b) {
  return a * b;
};

// double умножает только на два
var double = mul.bind(null, 2); // контекст фиксируем null, он не используется

alert( double(3) ); // = mul(2, 3) = 6
alert( double(4) ); // = mul(2, 4) = 8
alert( double(5) ); // = mul(2, 5) = 10
```

КОНТЕКСТ ФУНКЦИИ
ОБРАБОТЧИК СОБЫТИЙ

Когда функция используется как обработчик событий, this присваивается элементу с которого начинается событие (некоторые браузеры не следуют этому соглашению для слушателей добавленных динамически с помощью всех методов кроме addEventListener).

CODE SNIPPET

```
function bluify(e) {
  // Always true
  console.log(this === e.currentTarget);
  // true when currentTarget and target are the same object
  console.log(this === e.target);
  this.style.backgroundColor = '#A5D9F3';
}

var element = document.getElementById('somElement');

element.addEventListener('click', bluify, false);
```

CLOSURES

ЗАМЫКАНИЯ (CLOSURES)

Замыкание – это функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции и не в качестве её параметров.

ПРИМЕР ЗАМЫКАНИЯ

CODE SNIPPET

```
function makeAdder(x) {
  return function (y) {
    return x + y;
  };
};

var add5 = makeAdder(5);
var add10 = makeAdder(10);

console.log(add5(2));  // 7
console.log(add10(2)); // 12
```

МОДУЛИ

МОДУЛИ

Проблема отсутствия модулей в javascript может быть частично решена, за следующих паттернов:

1. Module pattern
2. Revealing module pattern
3. AMD
4. CommonJS

MODULE PATTERN

CODE SNIPPET

```
;(function() {
  function lodash(value) {
    // ...
  }

  var version = '2.4.1';

  function size(collection) {
    return Object.keys(collection).length;
  }

  lodash.size = size

  window._ = lodash;
}());
```

REVEALING MODULE PATTERN

CODE SNIPPET

```
var lodash = (function() {

  var version;
  function assignDefaults() { ... }

  return {
    defaults: function() {  }
  }

})();
```

!ASYNCHRONOUS MODULE DEFINITION (AMD)

CODE SNIPPET

```
//Calling define with a dependency array and a factory function
define(['dep1', 'dep2'], function (dep1, dep2) {
  //Define the module value by returning a value.
  return function () {};
});

// Or:
define(function (require) {
  var dep1 = require('dep1'),
    dep2 = require('dep2');
  return function () {};
});
```

COMMONJS

CODE SNIPPET

```
// In circle.js
const PI = Math.PI;

exports.area = function(r) { return PI * r * r;};

exports.circumference = function(r) { return 2 * PI * r;};

// In some file
const circle = require('./circle.js');
console.log( 'The area of a circle of radius 4 is ' +
circle.area(4));
```

ДЕСКРИПТОРЫ СВОЙСТВ

ДЕСКРИПТОРЫ, ГЕТТЕРЫ И СЕТТЕРЫ СВОЙСТВ

Object.defineProperty(obj, prop, descriptor)

- **value** – значение свойства, по умолчанию undefined
- **writable** – значение свойства можно менять, если true. По умолчанию false.
- **configurable** – если true, то свойство можно удалять, а также менять его в дальнейшем при помощи новых вызовов defineProperty. По умолчанию false.
- **enumerable** – если true, то свойство просматривается в цикле for..in и методе Object.keys(). По умолчанию false.
- **get** – функция, которая возвращает значение свойства. По умолчанию undefined.
- **set** – функция, которая записывает значение свойства. По умолчанию undefined.

НАСЛЕДОВАНИЕ

НАСЛЕДОВАНИЕ

1. Функциональное – метод с использованием наложения конструкторов
2. Прототипное наследование

СОЗДАНИЕ ОБЪЕКТОВ ЧЕРЕЗ "NEW"


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

var animal = new
Animal("ёжик");
```

CODE SNIPPET

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

  // в this пишем свойства,
методы
  this.name = name;
  this.canWalk = true;

  // return this;
}
```

ФУНКЦИОНАЛЬНОЕ НАСЛЕДОВАНИЕ

ФУНКЦИОНАЛЬНОЕ НАСЛЕДОВАНИЕ

Объявляется конструктор родителя Machine. В нём могут быть приватные (private), публичные (public) и защищённые (protected) свойства:

CODE SNIPPET

```
function Machine(params) {
  // локальные переменные и функции доступны только внутри Machine
  var privateProperty;

  // публичные доступны снаружи
  this.publicProperty = ...;

  // защищённые доступны внутри Machine и для потомков
  // мы договариваемся не трогать их снаружи
  this._protectedProperty = ...
}

var machine = new Machine(...)
machine.public();
```

ФУНКЦИОНАЛЬНОЕ НАСЛЕДОВАНИЕ

Для наследования конструктор потомка вызывает родителя в своём контексте через apply. После чего может добавить свои переменные и методы:

CODE SNIPPET

```
function CoffeeMachine(params) {
  // универсальный вызов с передачей любых
аргументов
  Machine.apply(this, arguments);

  this.coffeePublicProperty = ...
}

var coffeeMachine = new CoffeeMachine(...);
coffeeMachine.publicProperty();
coffeeMachine.coffeePublicProperty();
```

ФУНКЦИОНАЛЬНОЕ НАСЛЕДОВАНИЕ

В CoffeeMachine свойства, полученные от родителя, можно перезаписать своими. Но обычно требуется не заменить, а расширить метод родителя. Для этого он предварительно копируется в переменную:

CODE SNIPPET

```
function CoffeeMachine(params) {
  Machine.apply(this, arguments);

  var parentProtected = this._protectedProperty;
  this._protectedProperty = function(args) {
    parentProtected.apply(this, args); // (*)
    // ...
  };
}
```

ПРОТОТИПНОЕ НАСЛЕДОВАНИЕ

ПРОТОТИП ОБЪЕКТА

Если один объект имеет специальную ссылку __proto__ на другой объект, то при чтении свойства из него, если свойство отсутствует в самом объекте, оно ищется в объекте __proto__.

CODE SNIPPET

```
var animal = {
  eats: true
};
var rabbit = {
  jumps: true
};

rabbit.__proto__ = animal;

// в rabbit можно найти оба
свойства
alert(rabbit.jumps); // true
alert(rabbit.eats); // true
```

ПРОТОТИП ОБЪЕКТА МЕТОДЫ

- Чтение: Object.getPrototypeOf(obj)
- Запись: Object.setPrototypeOf(obj, proto)
- Создание объекта с прототипом: Object.create(proto, descriptors)
- obj.hasOwnProperty(prop) возвращает true, если свойство prop принадлежит самому объекту obj, иначе false.

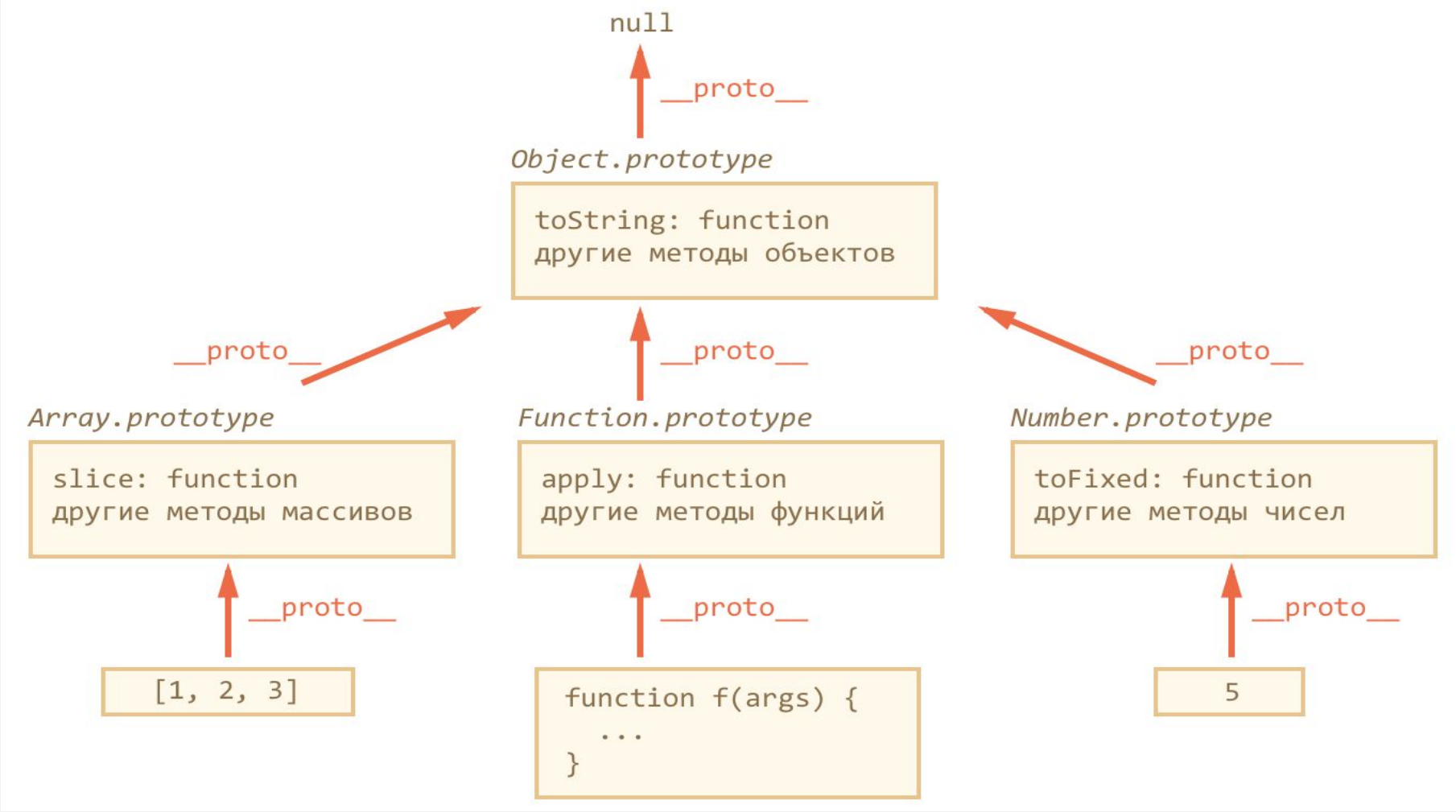
СВОЙСТВО F.PROTOTYPE

При создании объекта через new, в его прототип __proto__ записывается ссылка из prototype функции-конструктора.

CODE SNIPPET

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

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

Rabbit.prototype = animal;

var rabbit = new Rabbit("Кроль"); //  rabbit.__proto__ == animal

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

ВСТРОЕННЫЕ КЛАССЫ



СОЗДАНИЕ КЛАССА PARENT

CODE SNIPPET

```
// конструктор
function Animal(name) {
  this.name = name;
  this.speed = 0;
}

// методы в прототипе
Animal.prototype.run = function(speed) {
  this.speed += speed;
  alert( this.name + ' бежит, скорость ' + this.speed
);
};

Animal.prototype.stop = function() {
  this.speed = 0;
  alert( this.name + ' стоит' );
};
```

СОЗДАНИЕ КЛАССА CHILD

CODE SNIPPET

```
function Rabbit(name) {
  this.name = name;
  this.speed = 0;
}

// задаём наследование
Rabbit.prototype = Object.create(Animal.prototype);
Rabbit.prototype.constructor = Rabbit;

// и добавим свой метод (или методы...)
Rabbit.prototype.jump = function() { ... };
```

!ВАЖНО

CODE SNIPPET

```
// вместо Rabbit.prototype =
Object.create(Animal.prototype)
Rabbit.prototype = new Animal();
```

Но у этого подхода важный недостаток. Как правило мы не хотим создавать Animal, а хотим только унаследовать его методы!

Более того, на практике создание объекта может требовать обязательных аргументов, влиять на страницу в браузере, делать запросы к серверу и что-то ещё, чего мы хотели бы избежать. Поэтому рекомендуется использовать вариант с Object.create.

ПРОТОТИПНОЕ НАСЛЕДОВАНИЕ

CODE SNIPPET

```
// вызов конструктора родителя
function Rabbit(a, b) {
  Animal.apply(this, arguments);
}

// вызов метода родителя
Rabbit.prototype.run = function() {
  var result = Animal.prototype.run.apply(this, ...);
  // result -- результат вызова метода родителя
}
```

ПОЛЕЗНЫЕ ССЫЛКИ

1. https://developer.mozilla.org/bm/docs/Web/JavaScript
2. https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Details_of_the_Object_Model
3. https://learn.javascript.ru/
4. https://addyosmani.com/resources/essentialjsdesignpatterns/book/ (модули)
5. https://github.com/vvscode/js--interview-questions

CONTACTS:

Skype: live:toxablack
Email: charnou.anton@itechart-group.com

:iTechArt

ВОПРОСЫ?