IT-Academy.BG

Обектно-ориентирано програмиране в JavaScript

Димитър Митев

IT-Academy.BG

Обектно-ориентирано програмиране

- ООП подхода означава, че програмата/апликацията е конструирана като множество от обозрими обекти
  - Всеки обект си има конкретно предназначение
  - Всеки обект може да съдържа други обекти
  - Чрез обектите се описват реални обекти

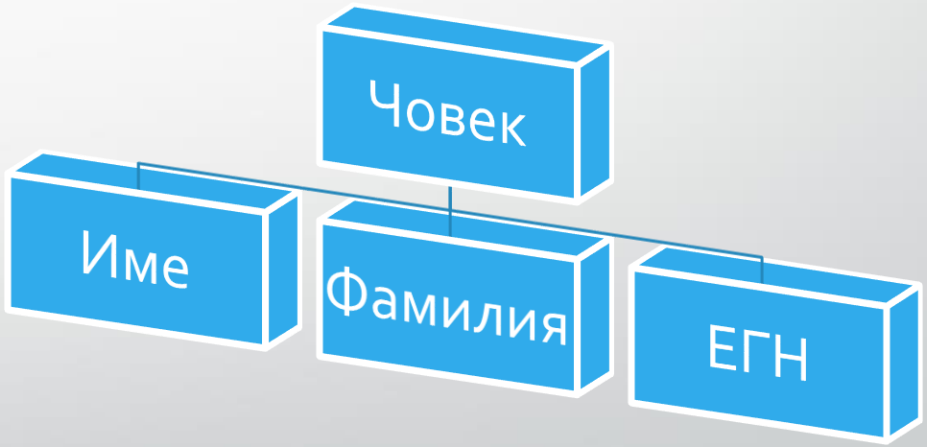
IT-Academy.BG

Прицнипи на обектно-ориентираното програмиране (ООП)

- Абстракция
- Наследяване
- Енкапсулация
- Полиморфизъм

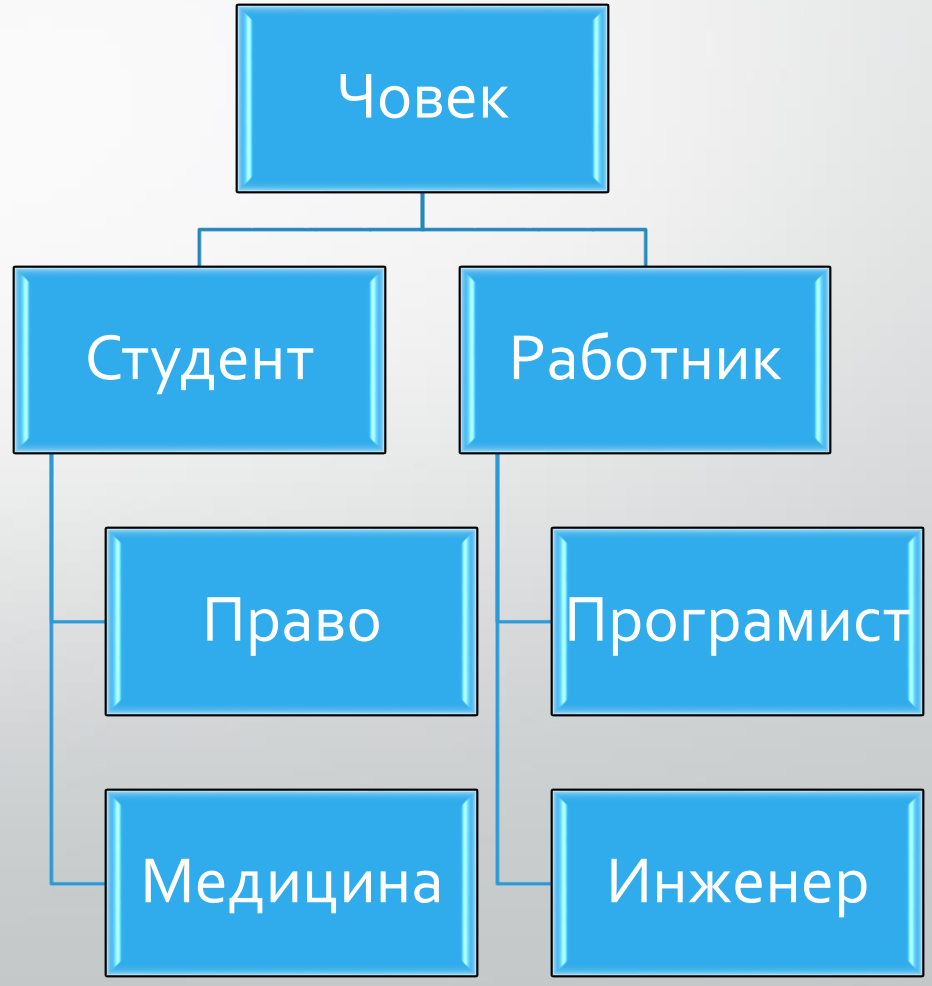
Абстракция

- Представя сложната реалност като прост модел
- Фокусира само върху това, което ни трябва без да се навлиза в конкретика
- Позволява ни да създаваме общовалидни модели, които да бъдат използвани за по-сложна логика



Наследяване

- Чрез наследяването споделяме общи характеристики между наследниците и техния родител
- Наследвяването ни помага да имаме високо ниво на абстракция
- Наследяването спестава повторяемо писане на код и поддръжката му



Енкапсулация

- Скриваме данни или логика в нашите модели
- Защитаваме нашата логика от намеса от външния свят
- Защитаваме чувствителна информация от достъп или възможност да бъде обработена
- „Защитно“ програмиране

Полиморфизъм

- Едно и също нещо може да приема различни форми
- 1 шаблон може да има различни имплементации

IT-Academy.BG

ООП в JavaScript

- JS не е типичен ООП език
- Основните принципи, които се използват в ООП в JS са
  - Наследяване
  - Енкапсулация
- JS не поддържа концепцията за класове
  - Класовете всъщност са функции, т.е имат собствен *scope*, но не са ясни класове като в езици като C# или JAVA
  - **В EcmaScript2015 вече съществува концепцията за класове**
- Mozilla Developers Network
- JS is sexy

- Терминология
  - **Class**-> дефинира характеристиките и действията на конкретен обект. Обикновено има име, изписано в PascalCase
  - **Property** -> определена характеристика на даден обект
  - **Method** -> определено действие на даден обект
  - **Constructor** -> метод, който се вика в момента на създаването на обекта. Обикновено има име като на класа и се изписва в PascalCase

IT-Academy.BG

ООП в JavaScript

- JavaScript е динамичен език
  - Няма типове, няма полиморфизъм
- JavaScript е силно експресивен език, т.е. 1 нещо може да бъде постигнато по много различни начини
- Поради своите особености JS поддържа различни видове ООП
  - Класическо/функционално, прототипно
  - Всеки тип си има своите предимства и недостатъци

Класическо наследяване в JS

Класическо ООП

- За да създава обекти, в JS се използва функции
  - *JS не поддържа ясна дефиниция за клас или конструктор*
  - *С въдеждането на EcmaScript 2015, вече съществува ясна дефиниция за клас и конструктор в JS*
- Функциите играят ролята на обектиния конструктор
  - Създаването/инициализирането на обекта се осъществява като се извика фунцкията с ключовата дума *new*

```
function Car(){ }
var mercedes = new Car(); // instance of Car
var tesla = new Car(); // another instance of Car
```

Създаване на обекти

- Когато се използва фунцкия като обектен конструктор, то тя се извиква за изпълнение чрез ключовата дума *new*

```
function Car(){ }
var mercedes = new Car(); // instance of Car
var tesla = new Car(); // another instance of Car
```

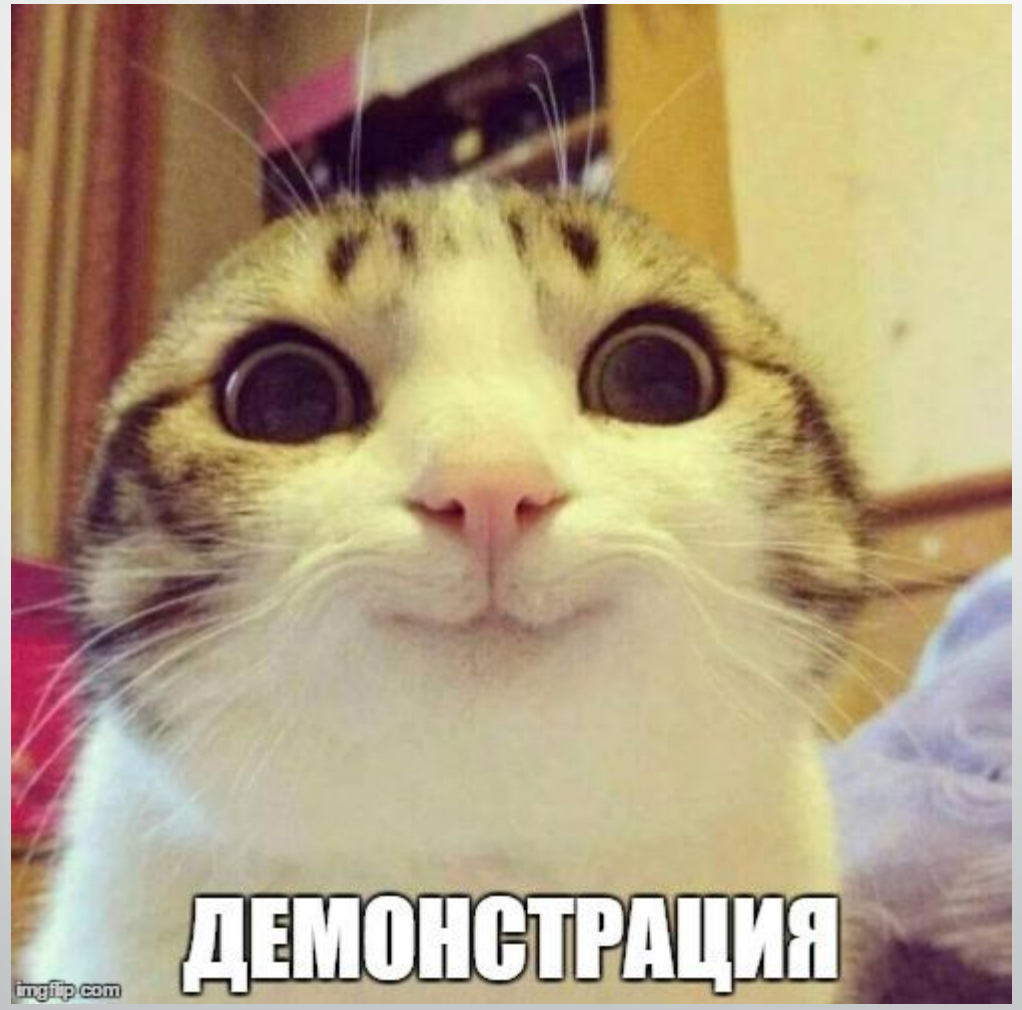
- Всяка инстанция е независима, т.е. всяка инстанция си има свое състояние и поведение
- Функцията конструктор може да приема параметри, за да задава текущото съсотяние на инстанцията

Създаване на обекти

- Фунцкия конструктор с параметри
  - Нормална функция с параматри, която се извиква с кл. дума *new*

```
function Car(make, model, year){
    this.make = make;
    this.model = model;
    this.year = year;
}
var mercedes = new Car('Mercedes', 'E320', 2006);
var tesla = new Car('Tesla', 'Model S', 2012);

console.log(Mercedes.model); // E320
console.log(tesla.year); // 2012
```

Създаване на обекти

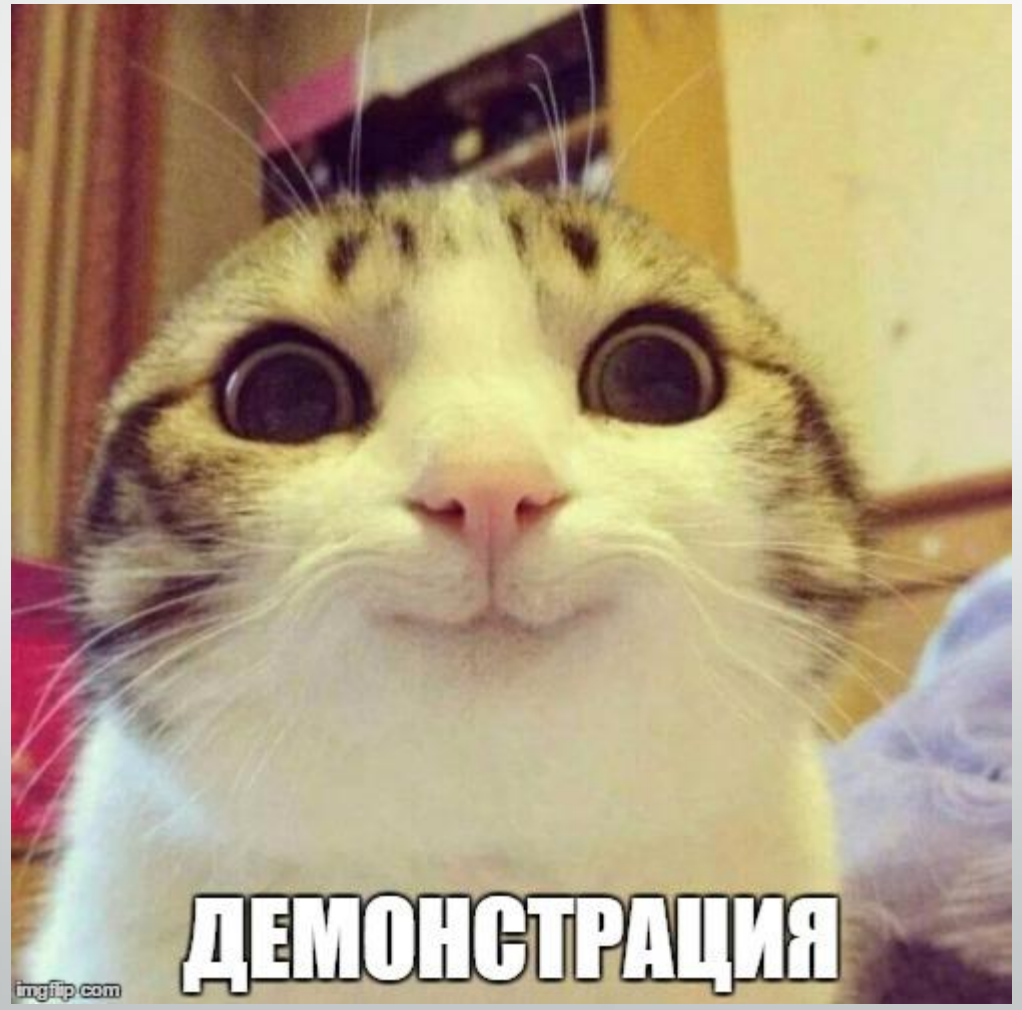


Прототип (prototype)

- JS е прототипно ориентиран език
  - Всеки обект има своя прототип
- Прототипите притежават характеристики, характерни общи за всички обекти
  - *Object* е обекта родител на всички обекти в JS
    - Напр. *Object* предоставя чрез своя *prototype* методи като *toString()* и *valueOf()* на всички обекти в JS
- Всички инстанции на обекта имат характеристиките, закачени на неговия протип

```
String.prototype.repeat = function(count) {
    var pattern = this;
    var str = '';

    if (!count) {
        return pattern;
    }

    for (var i = 0; i < count; i += 1) {
        str += pattern;
    }

    return str;
};

console.log('@'.repeat(5)); // @@@@@
```

Прототипи



Характеристики на обекта /object members/

- Обектите могат да имат различно състояние и това се получава благодарение на техните характеристики
- За да се дефинират характеристиките на обекта се използва ключовата дума *this*, което означава, че всяка нова инстанция си има конкретно състояния

this е текущия контекст на инстанцията на **Person**

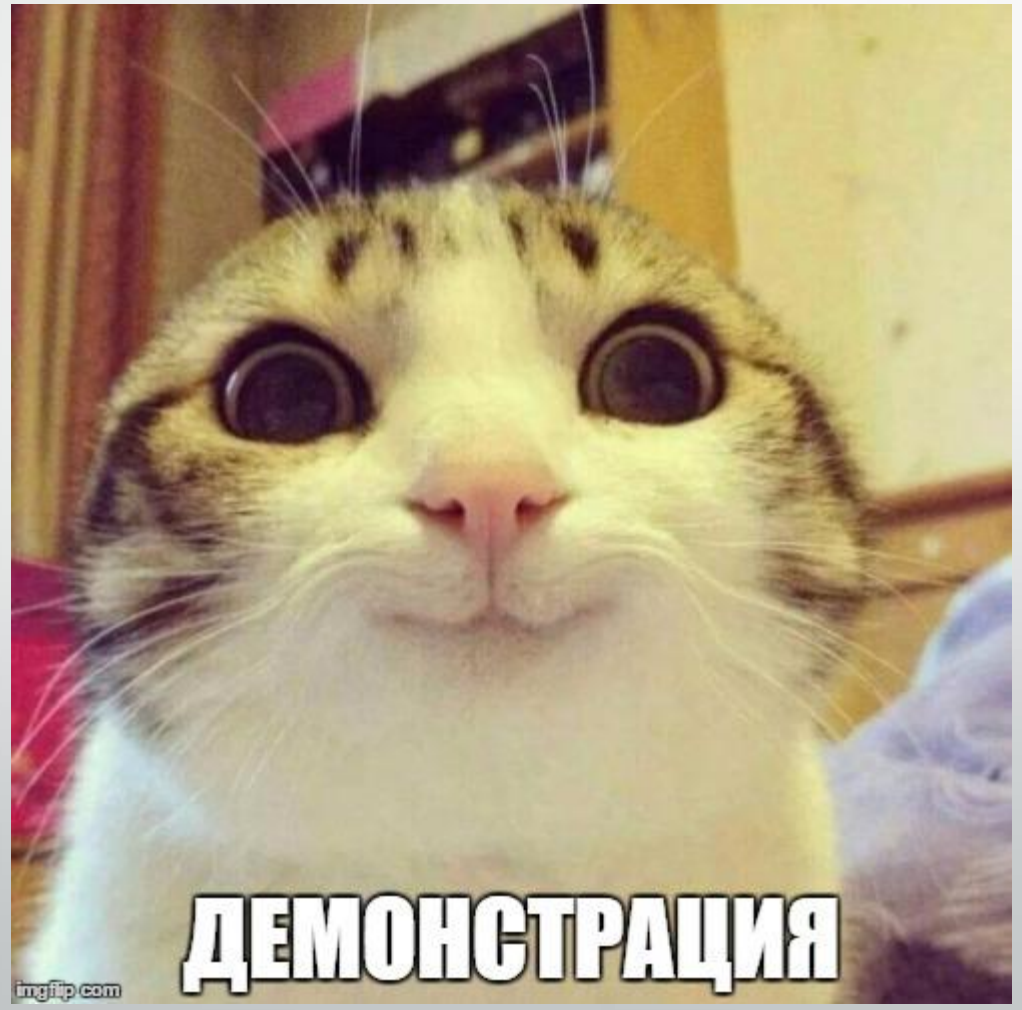
```
function Person(name, age) {
   this.name = name;
   this.age = age;
}
var mimi= new Person("Maria",18);
console.log(mimi.name);
```

Характеристики на обекта /object members/

- Характеристики на обекта могат да бъдат както променливи така и фунцкиции
- Функциите, които са характеристики на даден обект също се наричат и методи

```
function Person(name, age) {
   this.name = name;
   this.age = age;
   this.sayHello = function() {
      console.log('Hello! I'm ${this.name} and I'm ${this.age} years old.');
   }
}
var mimi= new Person("Maria",18);
mimi.sayHello();
```

Object members



Прикрепяне на методи към обект

- Закачането на методи към конструктора на обекта е „нож с 2 острието“
  - **Бавна** операция е
  - Всеки обект има фунцкия с 1 и съща логика, но въпреки това са различни инстанции

```
function Constr(){
   this.method = function(){
      // some code
   };
}
```

```
var x = new Constr();
var y = new Constr();

console.log (x.method === y.method);
```

false -> инстанциите са различни

IT-Academy.BG

Прикрепяне на методи към обект.

this

- Сходно с други езици
- Скриваме данни
- Лоша производителност

prototype

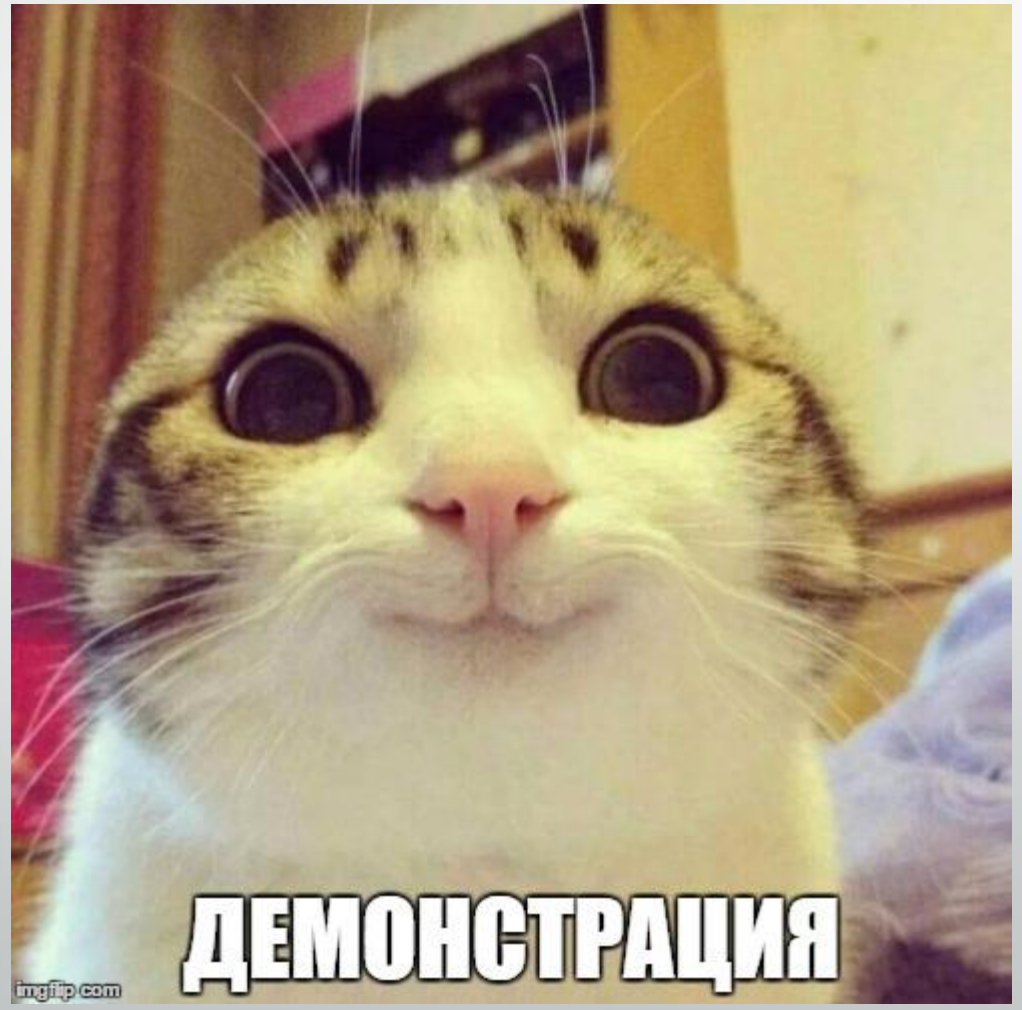
- Използваме JS така както е измислен да работи
- В JS данни се крият трудно, затова просто им слагаме префикс _ (долна черта)
- Добра производителност

JavaScript трбва да се третира като 1-вокласен език, а не като друг език.

Обектът *this*

- *this* е специален обект в JS; съществува навсякъде в JS, но има **различно** значение, според това къде и как се употребява
- *this* може да има 2 различни стойности
  - Родителския обхват /parent scope/
    - Стойността на *this* в съдържащия го *scope*
    - Ако никой от родителите не е обект, то *this* ще има стойност *window*
  - Конкретен обект
    - Когато е използван оператора *new*

Обектът *this*

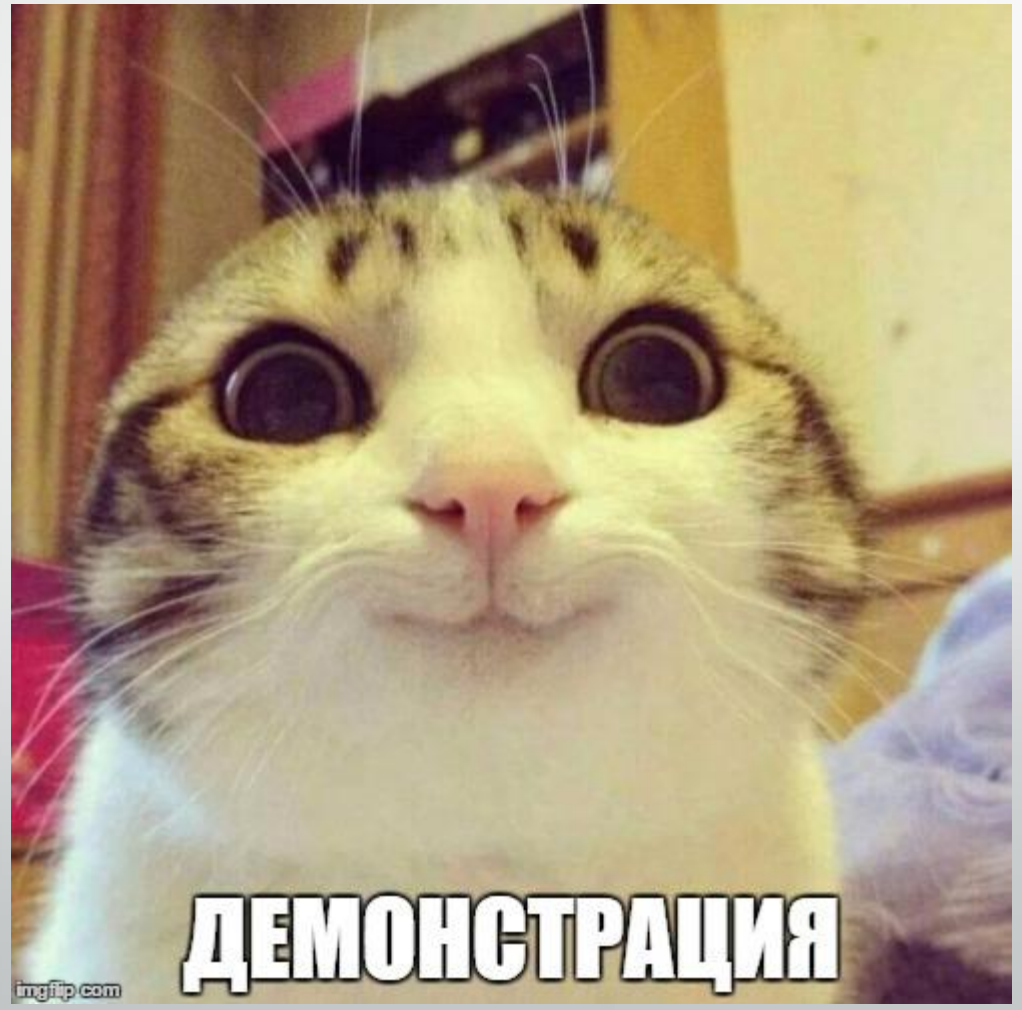


Модули

- Функциите конструктори могат да бъдат поставени в модули
  - Позволява по-добра абстракция на кода
  - Позволява да се скриват константни или функции
- JS поддържа 1вокласни функции, които могат да бъдат превърнати в модули

```
var Person = (function () {
    function Person(name) {
        //...
    }
    Person.prototype.walk = function (distance){ /*...*/ };

    return Person;
}());
```

Модули



Скрити функции в модул. Дефиниране и използване

- Когато функцията конструктор се използва в модул,то
  - Модула може да скрие в себе си функции
  - Фунцкията конструктор може да използва и достъпва скритите функции
- За да могат да бъдат изпозлвани въпросните скрити функции, те трябва да бъдат извикани посредством ключовата дума *call* или *apply*

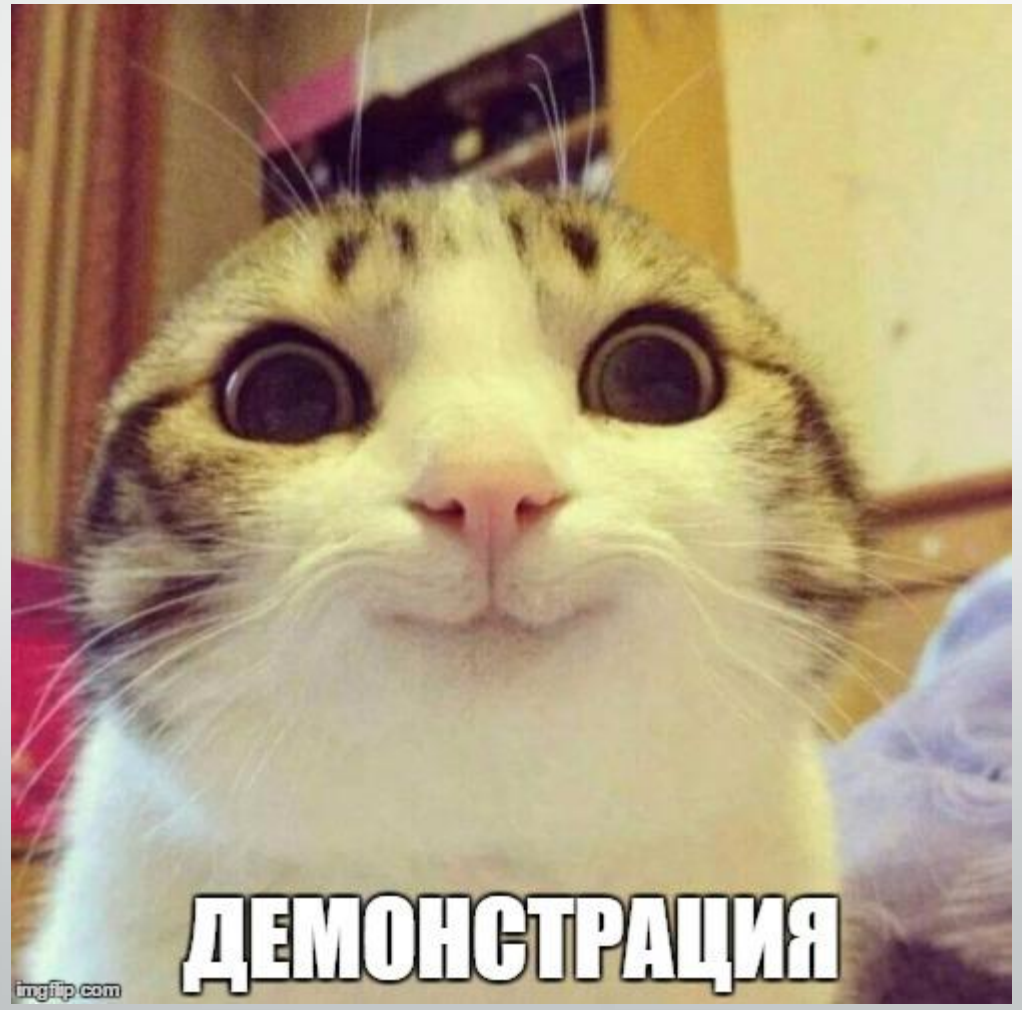
Скрити функции в модул. Дефиниране и използване

```
var Rect = (function () {
  function validatePosition() {
    //…
  }
  function Rect(x, y, width, height) {
    var isPositionValid = validatePosition.call(this);
    if (!isPositionValid) {
      throw new Error('Invalid Rect position');
    }
  }

  Rect.prototype = { /* … */};
  return Rect;
}());
```

Извън модула, функцията не съществува

Използва се **call()**, за да се извика ф-цията в/у **this** обекта на ф-цията конструктор

Скрити функции в модул



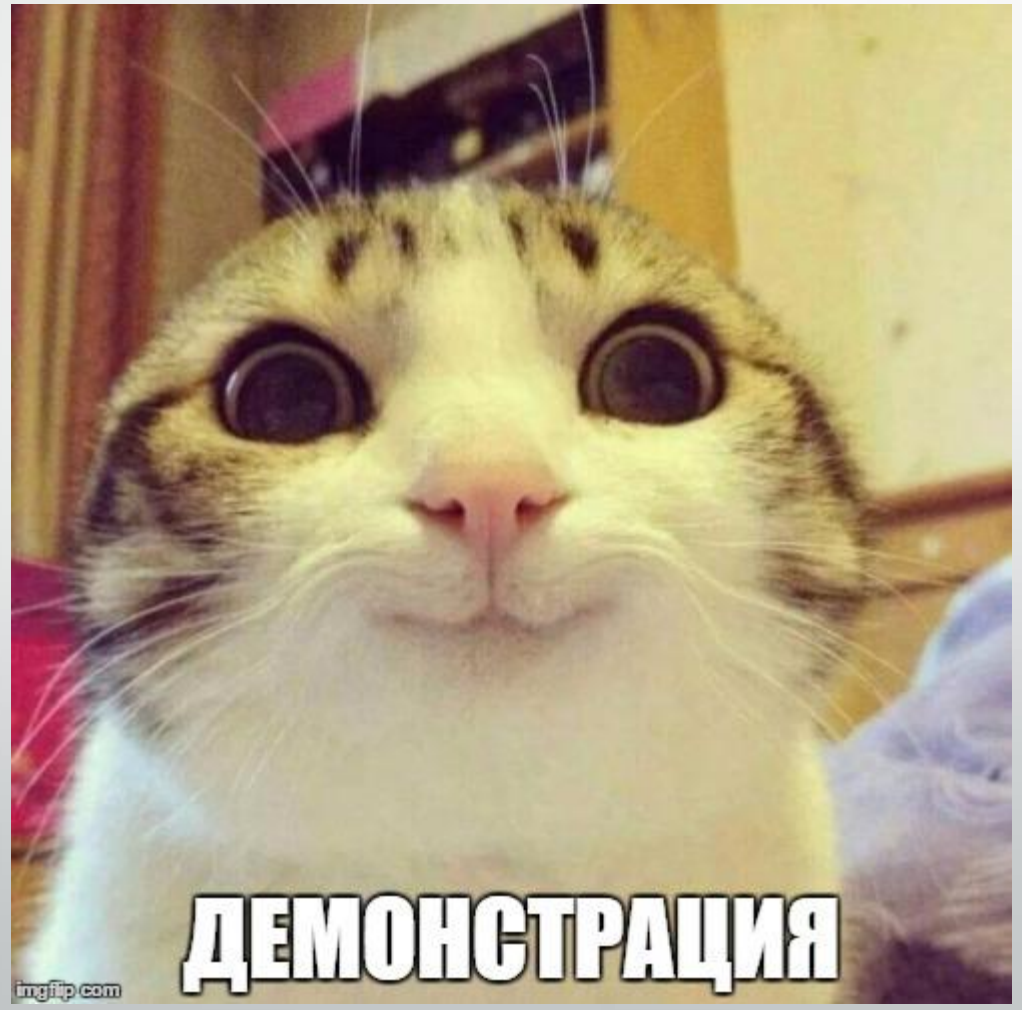
Наследяване в JS. Класическо ООП

- Наследяването е начин да се разшири функционалността/възможностите на даден обект, който се базира на друг обект
  - Напр. *Ученик* наследява *Човек*
- В JS наследяването се постига като на прототипа на наследника се задава за стойност прототипа на родителя

```
function Person(fname, lname) {}
function Student(fname, lname, grade) {}
Student.prototype = Person.prototype;
```

  - Всички инстанции на обекта *Student* ще бъдат съответно и от тип *Person* и ще имат функционалностите на *Person*

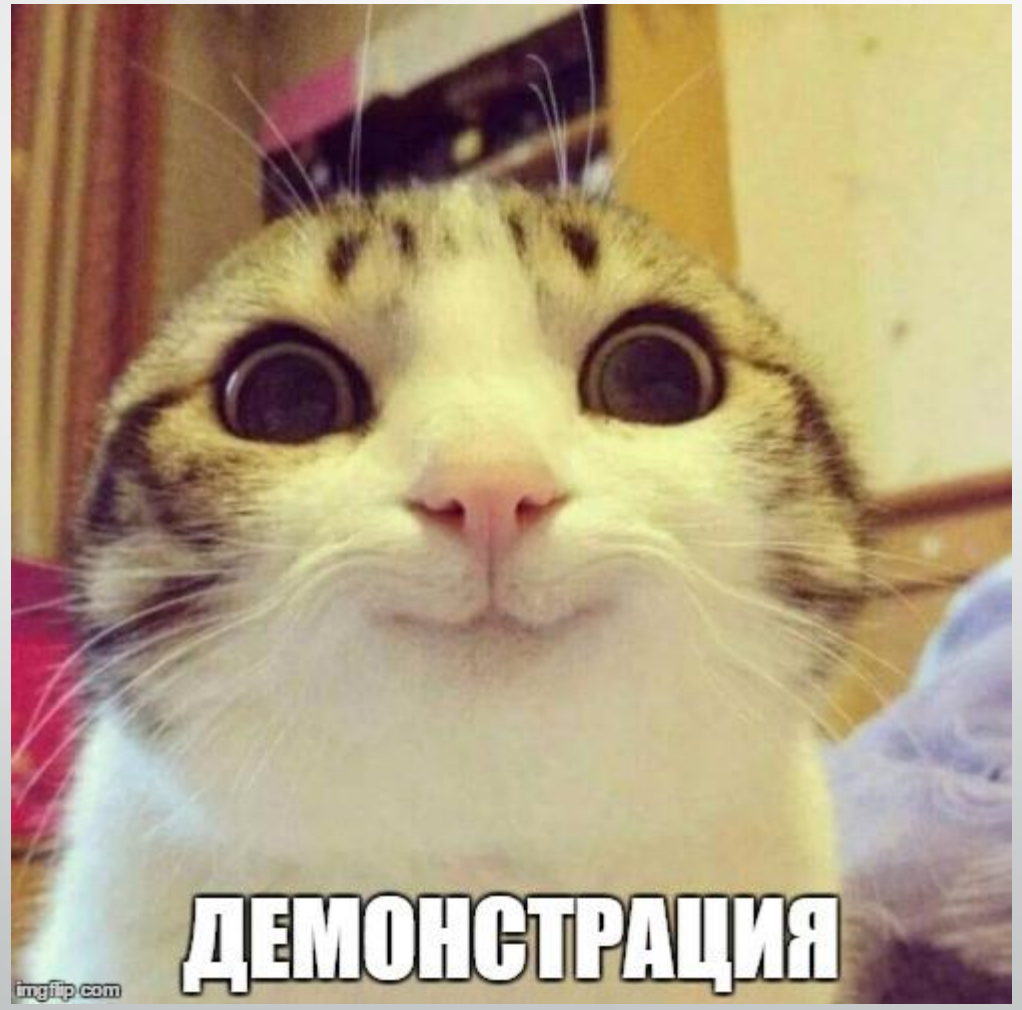
Наследяване в JS. Класическо ООП

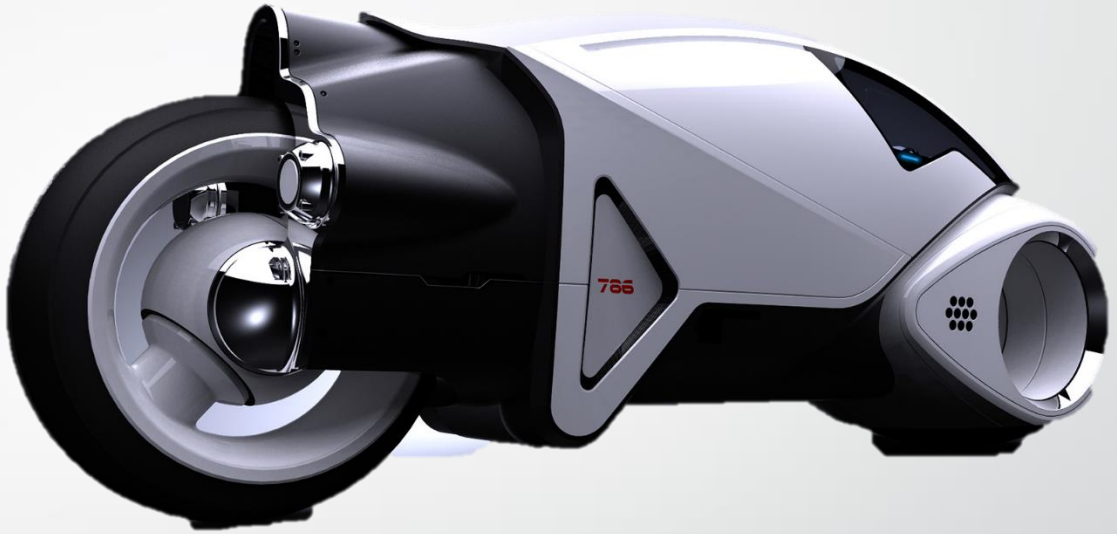


Извикване на родителски методи

- В JS няма директен начин за извикване на родителските методи, т.е. базовите методи
  - Функциите конструктори в действителност не знаят дали имат родител и кой е той
  - Поради тази причина извикването на родителеските методите се прави посредством *call()* или *apply()*, за да може да се подаде конкретния контекст

Извикване на родителски методи



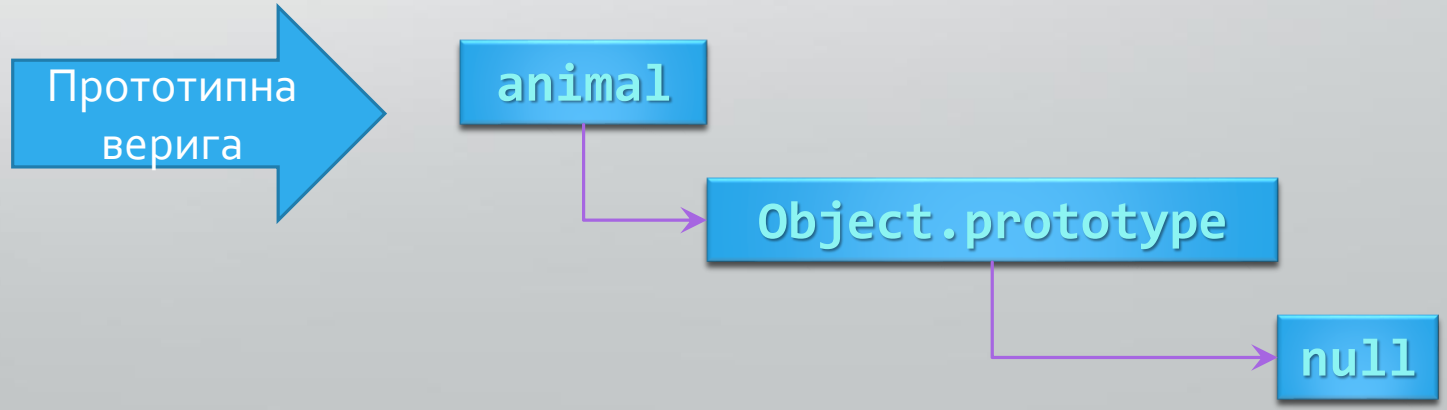


Прототипно наследяване в JS

Обектни прототипи

- Прототипът в действителност представлява обект
  - Прототипът осигурява конкретни характеристики на наследниците
- Всеки обект има своя прототип
  - *Object.prototype*
  - Това оформя т.нар. протипна веригра (*prototype chain*)
  - *Object* има за прототип *null*, което затваря протитпната верига

```
var animal = {
    /* properties and methods */
};
```

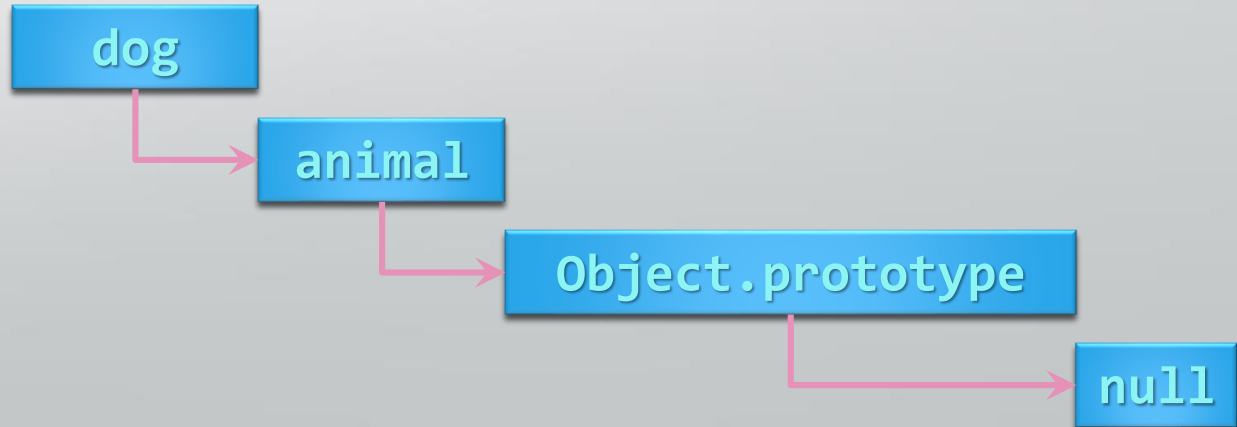


Настройване на прототипа на обект

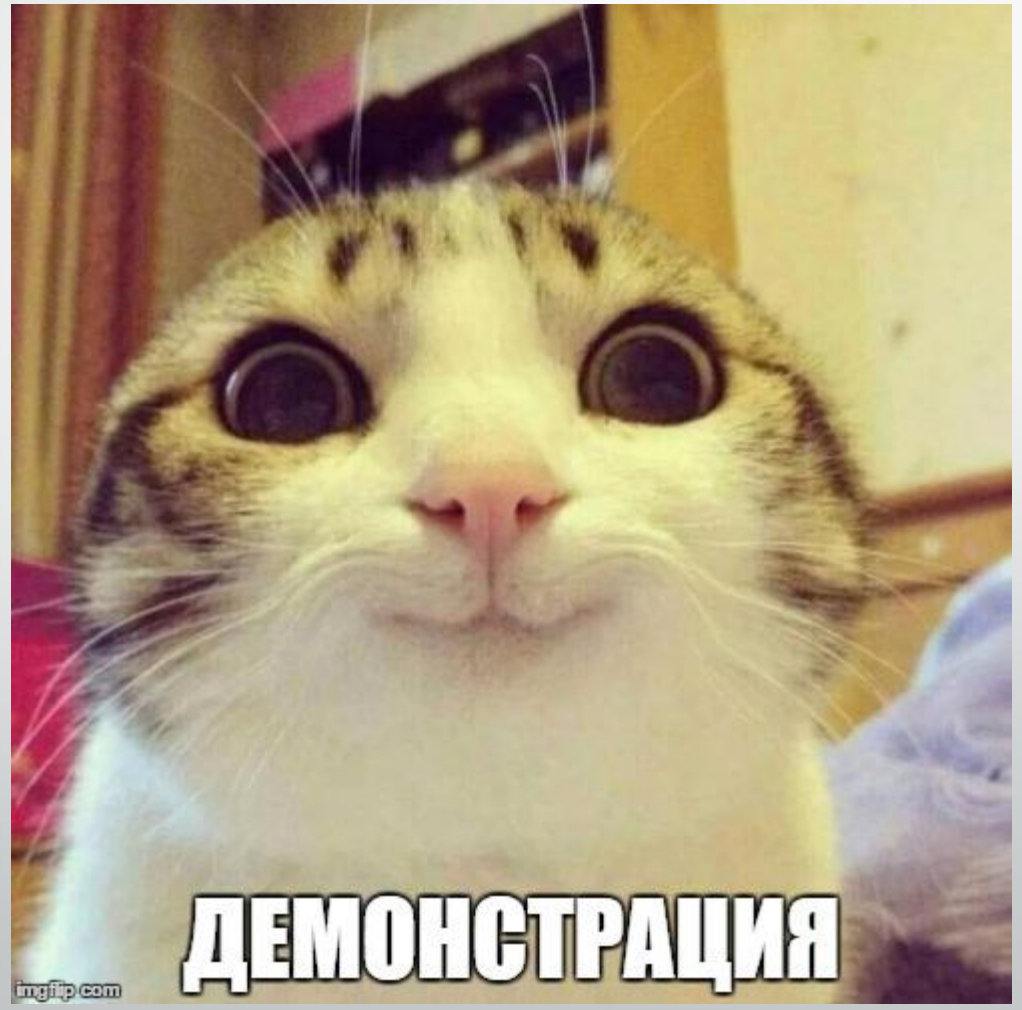
- Всеки обект в JS има характеристика (property) *__proto__*
  - Може да се използва, за да се достъпва/променя прототипа
  - Опасно е да се използва
- Когато нагласяме прототипа на обекта, обекта получава достъп до всички характерситики на прототипа
  - Чрез прототипната верига (prototype chain)
  - Това се нарича прототипно наследяване

Настройване на прототипа на обект

```
var animal = {
  makeNoise: function(){
    console.log('The ' + this.type +
                ' makes noise "'  + this.noise + '"');
  }
};
var dog = { type: 'dog', noise: 'Djaf' };
dog.__proto__ = animal;
dog.makeNoise(); // makeNoise() is from the prototype(animal)
```



Настройване на прототипа на обект чрез *__proto__*



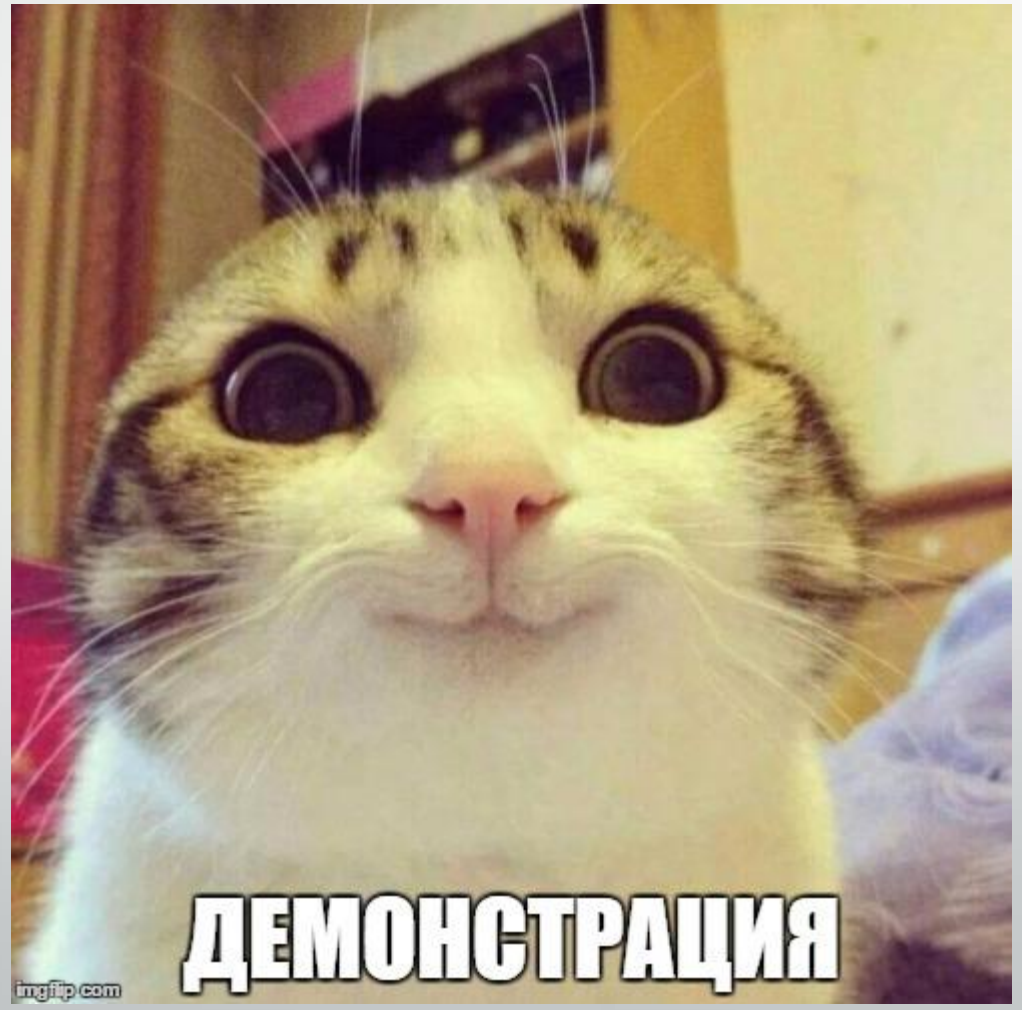
Настройване на прототипа на обект чрез *Object.create()*. По-добрия начин

- С въвеждането на EcmaScript5 през 2009 се въвежда и нова възможност за настройка на протипа на обект -> *Object.create()*

```
var animal = {
  /* properties and methods */
};
var dog = Object.create(animal);
dog.type = 'dog';
dog.noise = 'djaf';
```

```
var animal = {
  /* properties and methods */
};
var dog = {
  type: 'dog',
  noise: 'djaf'
};
Dog.__proto__ = animal
```

Object.create()



Работа с обектните прототипи

- *Object.create()* е добър начин, но е малко досадно да изреждаме и дефинираме всички характеристики на обекта чрез точкова нотация
  - Този проблем може да се реши чрез *Object.defineProperties(obj, props)*
    - *Дефинира множество характеристики на даден обект*
    - Може да се използва в IIFE, за да не замърсяваме глобалния *scope*
  - Този проблем може да се реши чрез *Object.defineProperty(obj, prop, descriptor)*

Работа с прототипа на обекта. Обекта *descriptor*

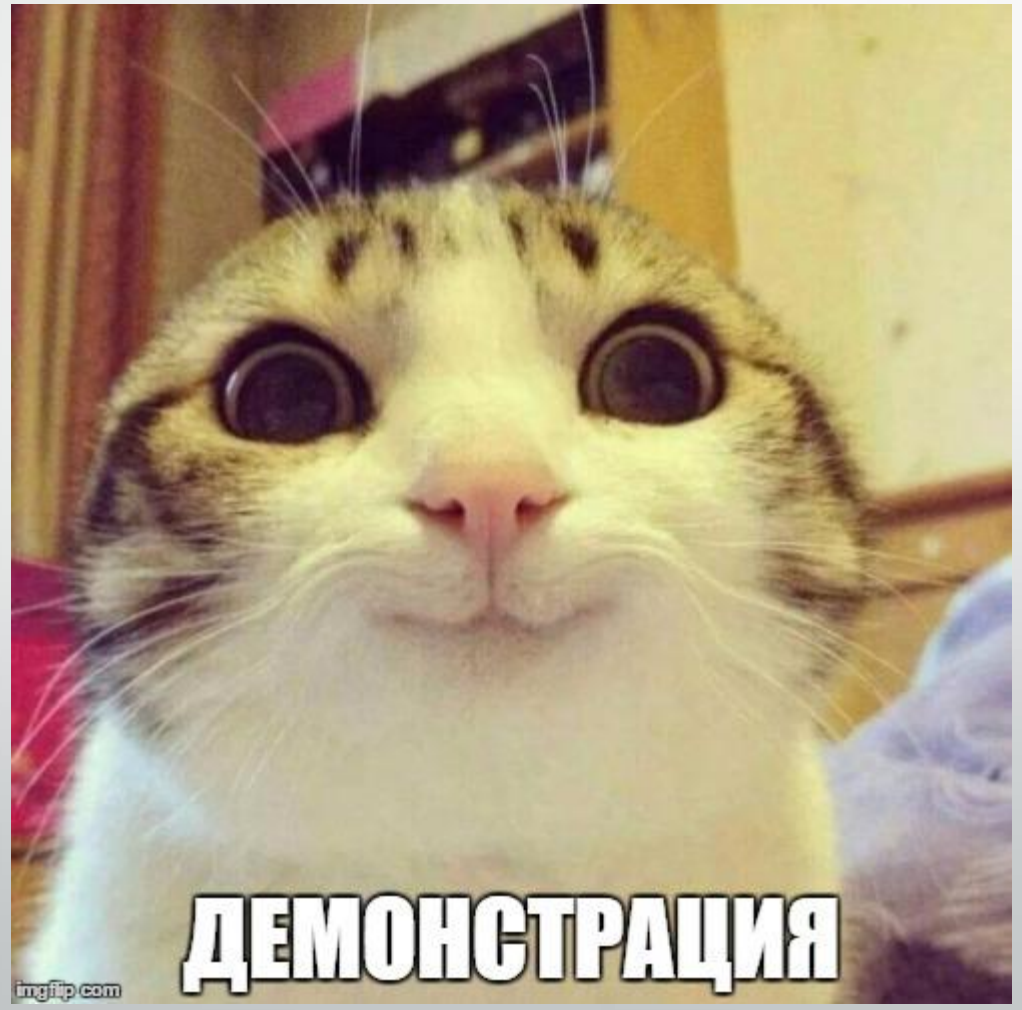
- *configurable* -> ако е *true*, то характерситиките на пропъртито могат да бъдат сменяни само в рамките на обекта. *false* по подразбиране
- *enumerable*-> ако е *true*, то въпростната характеристика е изброима. *true* по подразбиране
- *value*-> стойността на въпросната характеристика. *undefined* по подразбиране
- *writable* -> ако е *true*, то въпросната характеристиак може да бъде променяна чрез оператори за присвояване. *false* по подразбиране
- *get*-> функция, която позволява контрол върху това какво да получим като стойност, когато използваме конкретната характеристика
- *set*-> функция, която позволява контрол върху това какво да запишем като стойност, когато инициализираме конкретната характеристика

Работа с обектните прототипи

```
var dog = Object.defineProperties(animal, {
    type: {
      value: 'dog'
    },
    noise: {
      value: 'djaf'
    },
    bark: {
      value: function (){
        console.log('Bark, Bark');
      }
    }
  });
  return dog;
});
```

```
var dog = (function (parent) {
  var dog;
  /* hidden methods */
  dog = Object.defineProperties(parent, {
    type: {
      value: 'dog'
    },
    noise: {
      value: 'djaf'
    },
    bark: {
      value: function (){
        console.log('Bark, Bark');
      }
    }
  });
  return dog;
});
```

Object.defineProperties ()

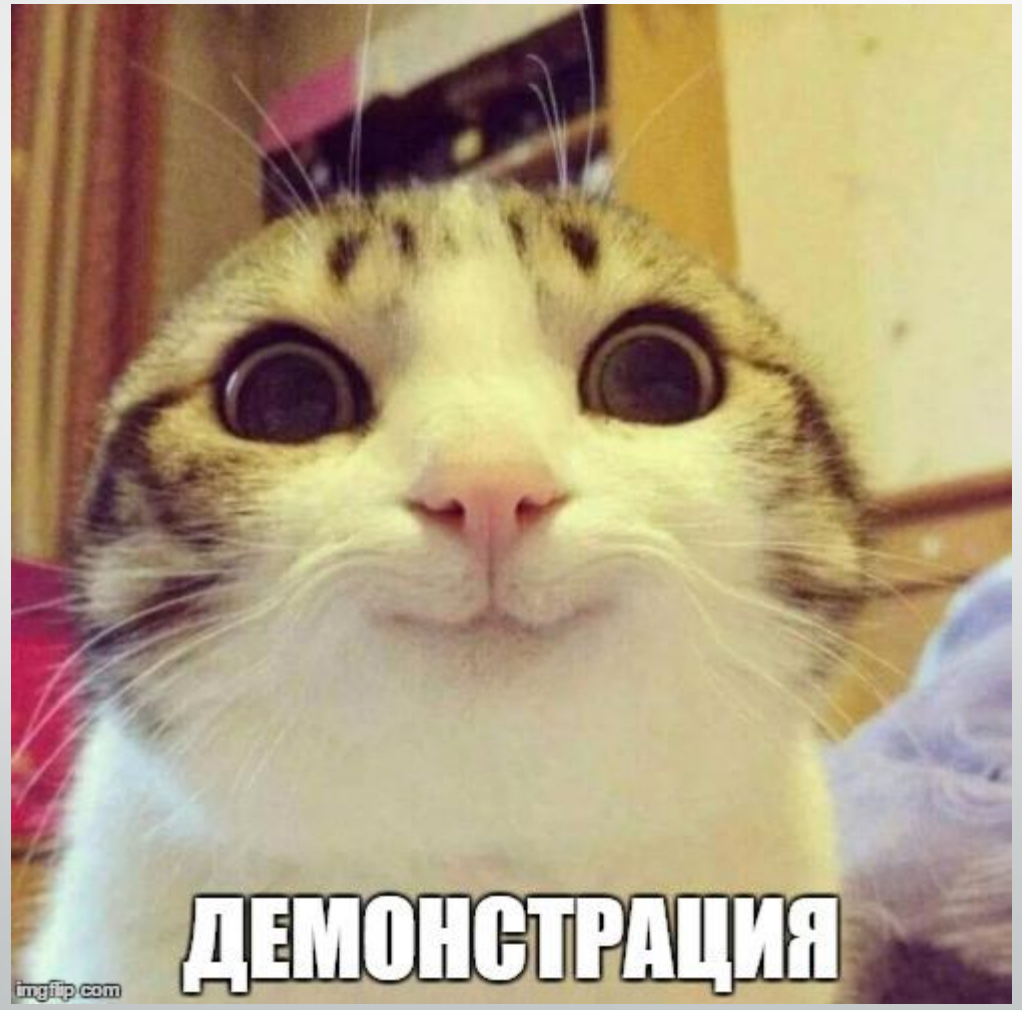


IT-Academy.BG

Преизползване на родителски методи

- Преизползването на родителските методи е доста важно в ООП
  - Спестява писане на повтаряем код
  - Прави поддръжката по-лесна
- В JS това се прави посредством *call()* или *apply()*

Преизползване на родителски методи



IT-Academy.BG

Изключения. Грешки. Управление на грешки и изключения

- Изключенията /exceptions/ са специални обекти в JS, които държат информация относно възникнали грешки
- Изключенията са правилния начин да се обработват възникнали грешки по време на изпълнението на програмата
- Вградени грешки в JS
  - SyntaxError
  - ReferenceError
  - JS Errors in Mozilla Developers Network

IT-Academy.BG

Обработка на изключения. /Exception handling/

- Обработка на изключения
  - Прихващане на изключение
  - Решение на проблема
  - Продължава изпълнението на кода
- Обработката на изключения позволява да се прихванат грешките и да се обработят конкретно без да се прекратя изпълнението на кода

Обработка на изключения. /Exception handling/

- Обработка на изключения в JS е посредством *try-catch* конструкцията

```
try {
    // code that can throw an exception
} catch (ex) {
    // if the above code throws an exception this code is
    // executed and ex holds the info about the exception
}
```

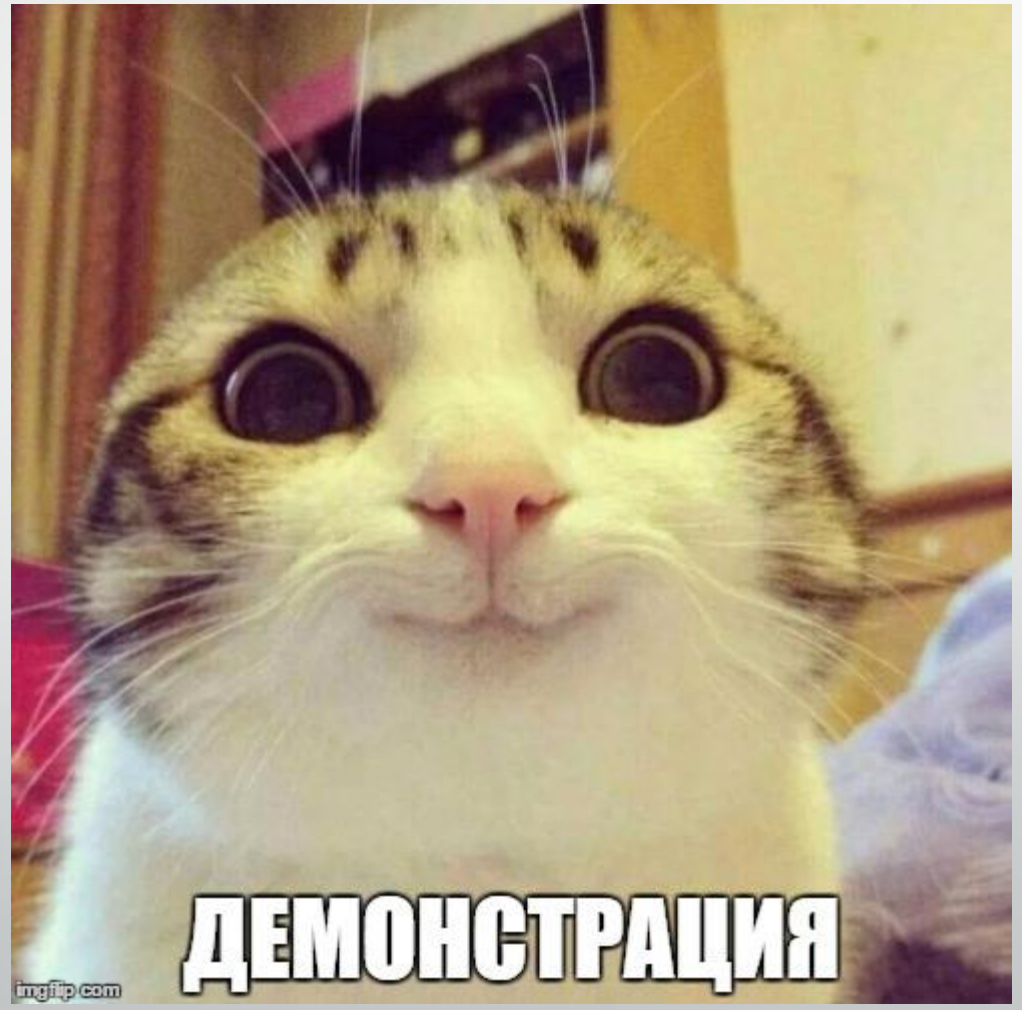
- В тялото на try блока се изпълнява код, който е възможен да стигне до изключение /грешка/
- Възникналата грешка се обработва в catch блока

IT-Academy.BG

Обработка на изключения. /Exception handling/

- Всеки *try-catch* съдържа само един try и само 1 catch
  - Ако очаквате специфична грешка, то трябва да проверите типа на грешката вътрв в catch блока
- Обекта на грешката съдържа информация за самата грешка
  - Нейния тип
  - Съобщението на грешката

Обработка на изключения



Създаване на изключения

- Изключение могат да бъдат създавани посредством конструктор
  - Конструктора има опционален параметър за съобщение, ако се пропусне, то получава стойност празен стринг

```
var typeException = new TypeError([message]);
var rangeException = new RangeError([message]);
```

  - Изкл‘чение могат да бъдат хвърляни чрез ключовата дума *throw*

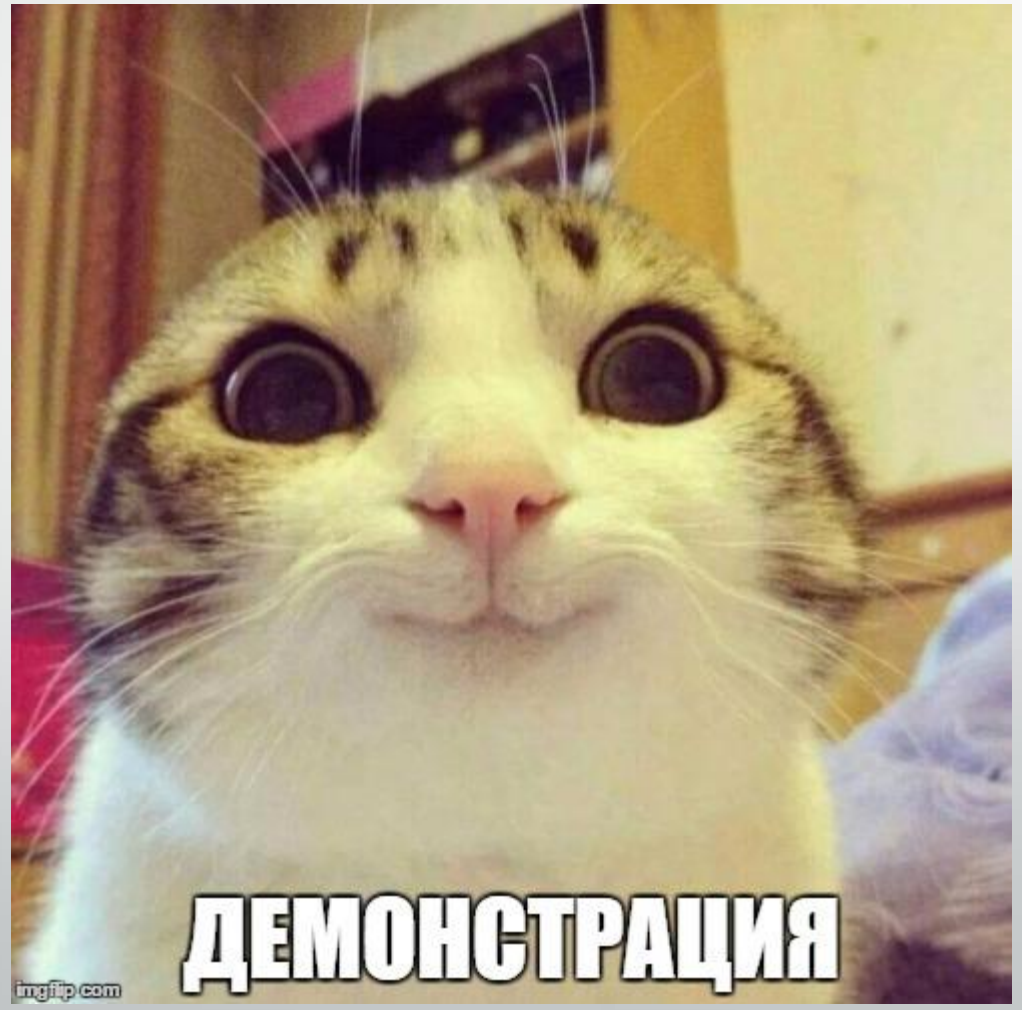
```
var typeEx = new TypeError("Not correct use of an object");
throw typeEx;
```

Собствени изключние

- Може да се създават и собствени изключения
- Използва се ключовата дума *throw* и след нея се задава, това което се вдига като изключение
- Може да се хвърли просто нова инстанция на Error обекта със съответно съобщение

```
throw new Error('Name is too short');
throw 'I have a custom erro';
throw { msg: 'Error', type: 'CustomError' };
```

Собствени изключения



IT-Academy.BG

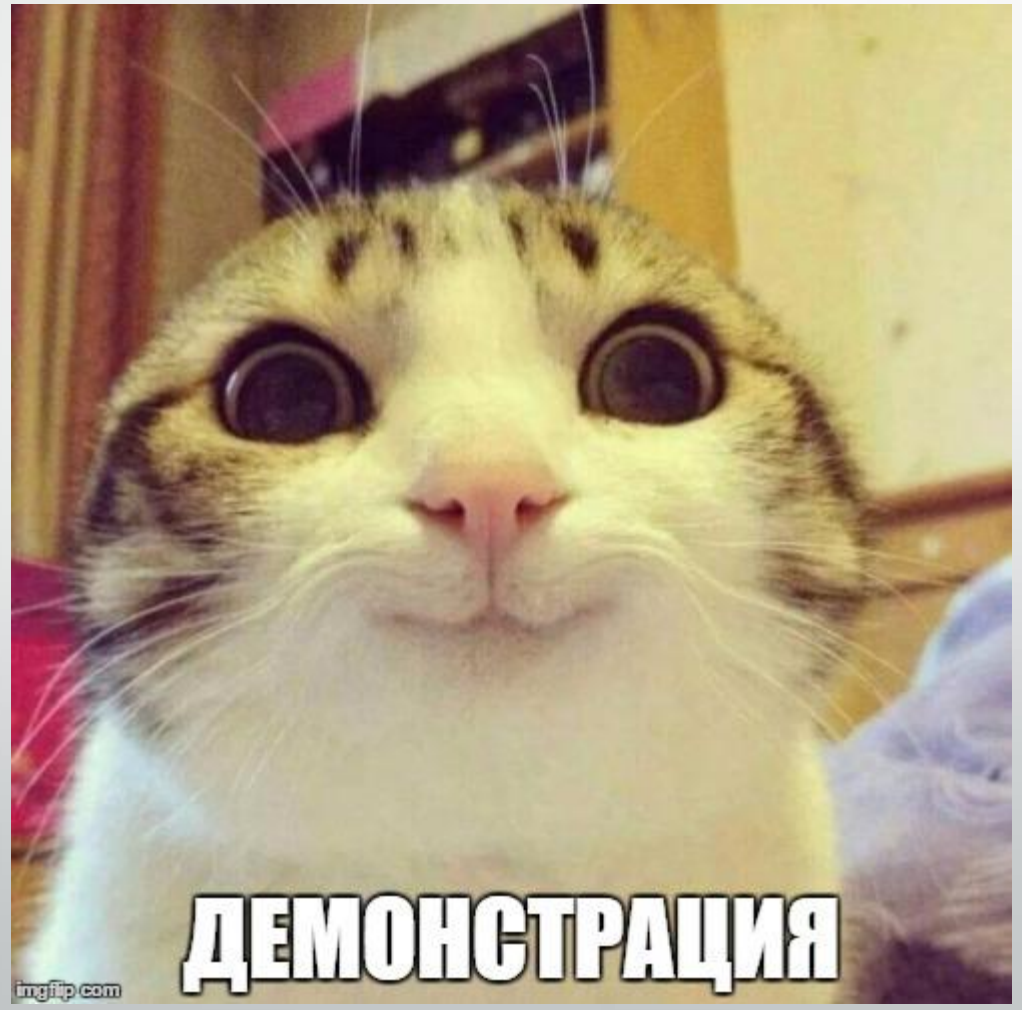
ООП в EcmaScript 2015

- ES2015 е отскоро въведен и не се поддържа от малко по-стари системи
- Добро решение за JS, особено по-лесен за ООП
- Ясни дефиниции на класове и обекти
- Syntax sugar

Обект в ES2015

- Абстракцията на обект от реалността в JS, при ES2015 вече се нарича клас и има ясна дефиниция за class
- Класът има конструктор
- Има съкратено изписване на getter-и и setter-и
- С ES2015 пишем доста по-лесно класическо ООП

ООП с EcmaScript2015




НЯКОЙ ИМА ЛИ
ВЪПРОСИ?

IT-Academy.BG

Домашна работа

1. Напишете JS ООП ориентиран код за фирма
   - Фирмата има име и брой служители
   - Във фирмата могат да бъдат назначавани или уволнявани хора
   - Всеки служител има име, позиция и може да се представи
   - *Използвайте какъвто ООП подход намерите за добре и за удобен