ПРОТОТИПНОЕ НАСЛЕДОВАНИЕ

Tinkoff.ru

- Оператор `new`
- Свойство `prototype`
- Прототипное наследование
- Классы ES6
- Пожалуйста, хватит примеров из Звездных Воин.

ДОПУСТИМ, ЛЕВ

```
const simba = {
  name: 'Симба'
};
```

```
const simba = {
  name: 'Симба',
  growl () {
    console.log(`${this.name} рычит!`);
  }
};
```

ДОПУСТИМ, ДВА ЛЬВА

```
const simba = {
  name: 'Симба',
  growl () {
    console.log(`${this.name} рычит!`);
  }
};
```

```
const mufasa = {
  name: 'Муфаса',
  growl () {
    console.log(`${this.name} рычит!`);
  }
};
```

КАК СДЕЛАТЬ МНОГО ЛЬВОВ?

ФУНКЦИИ-КОНСТРУКТОРЫ

```
function Lion (name) {
  this.name = name;
  this.growl = function () {
    console.log(`${this.name} рычит!`);
  }
}

const simba = new Lion('Симба');
const mufasa = new Lion('Муфаса');
```

Функция-конструктор - это любая функция, кроме стрелочной, вызываемая через `new`.

Создание объектов через "new"

`new`

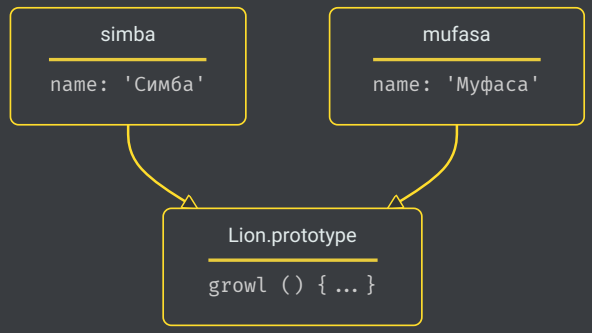
1. Создает новый объект
2. ???
3. Вызывает конструктор в контексте этого объекта
4. Возвращает объект

```
function Lion (name) {
  this.name = name;
  this.growl = function () {
    console.log(`${this.name} рычит!`);
  }
}

const simba = new Lion('Симба');
const mufasa = new Lion('Муфаса');

simba.growl(); // Симба рычит!
```

Что не так?

У каждого льва свой экземпляр функции `growl`.

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

Lion.prototype.growl = function () {
  console.log(`${this.name} рычит!`);
}

const simba = new Lion('Симба');
const mufasa = new Lion('Муфаса');

simba.growl(); // Симба рычит!
```



Общие для всех объектов методы можно вынести в прототип.

`new`

1. Создает новый объект
2. Копирует ссылку на объект `prototype` в прототип объекта
3. Вызывает конструктор в контексте этого объекта
4. Возвращает объект

1. Создать новый объект

```
const newOperator = function (constructor, ...args) {
  const object = {};

  Object.setPrototypeOf(object, constructor.prototype);

  constructor.call(object, ...args);

  return object;
}
```

Воспроизведение оператора new через функцию.

2. Задать ему прототип

```
const newOperator = function (constructor, ...args) {
  const object = {};

  Object.setPrototypeOf(object, constructor.prototype);

  constructor.call(object, ...args);

  return object;
}
```

3. Вызвать конструктор

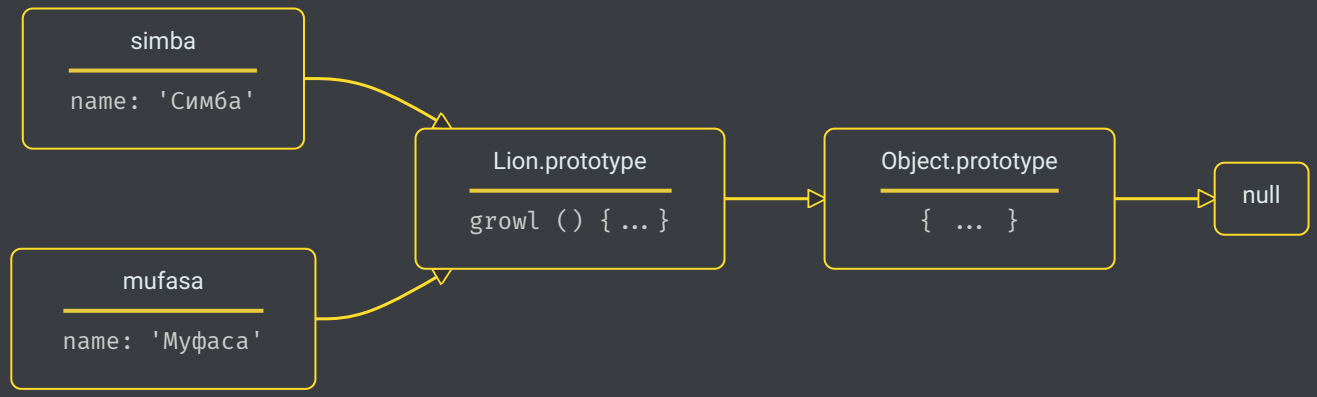
```
const newOperator = function (constructor, ...args) {
  const object = {};

  Object.setPrototypeOf(object, constructor.prototype);

  constructor.call(object, ...args);

  return object;
}
```

4. Вернуть готовый объект

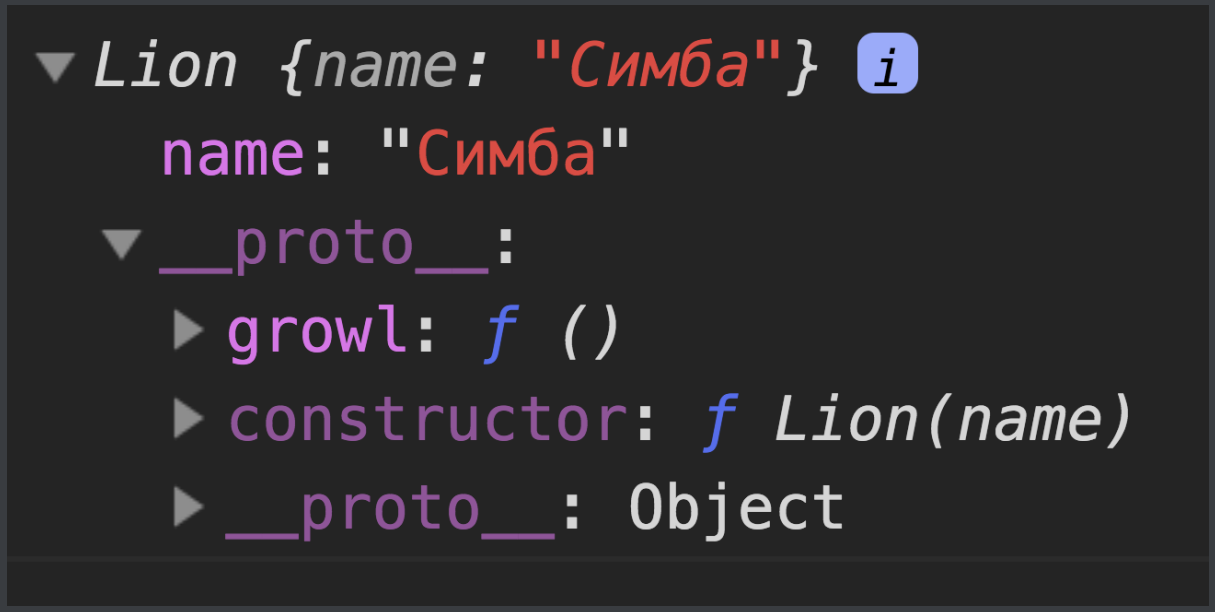
```
const newOperator = function (constructor, ...args) {
  const object = {};

  Object.setPrototypeOf(object, constructor.prototype);

  constructor.call(object, ...args);

  return object;
}
```

ЦЕПОЧКА ПРОТОТИПОВ

```
const simba = new Lion('Симба');
const mufasa = new Lion('Муфаса');
```



Сначала собственные свойства, потом прототип, потом прототип прототипа и так далее, пока не доходит до Object.prototype




```
simba.__proto__ === Lion.prototype;
```

`__proto__` у экземпляра и `prototype` у конструктора ссылаются на один и тот же объект.


```
simba.__proto__.__proto__ === Object.prototype;
```

У прототипа тоже есть прототип

```
simba.__proto__.__proto__.__proto__ === null;
```

Прототип у Object.prototype === null

```
const array = [];
array.filter === Array.prototype.filter; // true

const map = new Map();
map.forEach === Map.prototype.forEach; // true

const date = new Date();
date.getDay === Date.prototype.getDay; // true
```

Все методы, которые есть у массивов, объектов, дат, регэкспов и т.д. хранятся в прототипах соответствующих конструкторов.

```
const foo = {
  bar: 1,
  __proto__: {
    bar: 2
  },
};

foo.bar; // ???
```

```
const foo = {
  bar: 1,
  __proto__: {
    bar: 2
  },
};

foo.bar; // 1
```

```
const foo = {
  bar: 1,
  __proto__: {
    bar: 2
  },
};

delete foo.bar;
foo.bar; // ???
```

```
const foo = {
  bar: 1,
  __proto__: {
    bar: 2
  },
};

delete foo.bar;
foo.bar; // 2
```

```
const foo = {
  bar: 1,
  __proto__: {
    bar: 2
  },
};

delete foo.bar;
delete foo.__proto__.bar;
foo.bar; // ???
```

```
const foo = {
  bar: 1,
  __proto__: {
    bar: 2
  },
};

delete foo.bar;
delete foo.__proto__.bar;
foo.bar; // undefined
```

```
const simba = new Lion('Симба');
const mufasa = new Lion('Муфаса');

Lion.prototype.introduce = function (name) {
  console.log(`Привет, я ${this.name}`);
};

simba.introduce(); // Привет, я Симба
mufasa.introduce(); // Привет, я Муфаса
```

Прототип можно расширять динамически. Свойства и методы, добавленные в прототип, будут доступны даже у уже созданных экземпляров.

```
if (!Array.prototype.hasOwnProperty('flat')) {
  Array.prototype.flat = function (depth) {
    // code
  }
}

[[1], [2]].flat(); // [1, 2]
```

Эта возможность часто используется для так называемого полифиллинга.

```
class Lion {
  constructor (name) {
    this.name = name;
  }

  growl () {
    console.log(`${this.name} рычит!`);
  }
}

const simba = new Lion('Симба');
```

Классы ES6

КОНСТРУКТОР

```
class Lion {
  constructor (name) {
    this.name = name;
  }

  growl () {
    console.log(`${this.name} рычит!`);
  }
}
```

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

Lion.prototype.growl = function () {
  console.log(`${this.name} рычит!`);
}
```

ПРОТОТИП

```
class Lion {
  constructor (name) {
    this.name = name;
  }

  growl () {
    console.log(`${this.name} рычит!`);
  }
}
```

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

Lion.prototype.growl = function () {
  console.log(`${this.name} рычит!`);
}
```

СТАТИЧЕСКИЕ МЕТОДЫ

```
class Lion {
  constructor (name) {
    this.name = name;
  }

  static isLion (lion) {
    return lion instanceof Lion;
  }
}

Lion.isLion(new Lion()); // true
```

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

Lion.isLion = function (lion) {
  return lion instanceof Lion;
};

Lion.isLion(new Lion()); // true
```

instanceof

```
function Lion () {};
const lion = new Lion();

lion instanceof Lion; // true

// Потому что
lion.__proto__ === Lion.prototype;
```

```
function Lion () {};
const lion = new Lion();

lion instanceof Object; // true

// Потому что
lion.__proto__.__proto__ === Object.prototype;
```

```
const someProto = {};
function foo () {}
foo.prototype = someProto;

const bar = { __proto__: someProto };

// true (хотя не имеет отношения к foo)
bar instanceof foo;

// Потому что
bar.__proto__ === foo.prototype;
```

ГЕТТЕРЫ И СЕТТЕРЫ

```
class Lion {
  constructor (name) {
    this.name = name;
  }

  get introduction () {
    return `Привет, я ${this.name}`;
  }
}

const simba = new Lion('Симба');
simba.introduction; // Привет, я Симба
```

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

Object.defineProperty(
  Lion.prototype,
  'introduction',
  {
    get () {
      return `Привет, я ${this.name}`;
    }
  }
)

const simba = new Lion('Симба');
simba.introduction; // Привет, я Симба
```

"Настоящие" классы нельзя вызвать без `new`

```
class Lion { ... }

const simba = Lion('Симба');
// TypeError!
```

```
function Lion (name) { ... }

const simba = Lion('Симба');
// Нууу... ладно...
```

```
function Lion (name) {
  if (/* ??? */) {
    throw new Error("Lion needs to be called with the new keyword");
  }

  // code
}
```

Какую проверку можно сделать в конструкторе, чтобы убедиться, что он вызван через `new`?

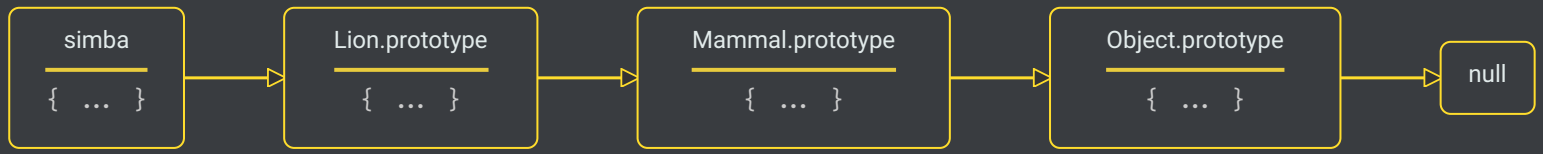
```
function Lion (name) {
  if (!(this instanceof Lion)) {
    throw new Error("Lion needs to be called with the new keyword");
  }

  // code
}
```

НАСЛЕДОВАНИЕ

```
class Mammal {
  constructor (name) {
    this.name = name;
  }

  growHair () {
    console.log(`${this.name} говорит: у меня растет шерсть`);
  }
}

class Lion extends Mammal {
  constructor (name, gender) {
    super(name);
    this.gender = gender;
  }
}

const simba = new Lion('Симба', '🚹');
simba.growHair(); // Симба говорит: у меня растет шерсть
```

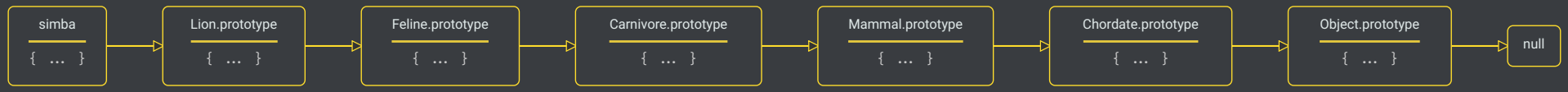
ЦЕПОЧКА ПРОТОТИПОВ

```
class Mammal { ... }
class Lion extends Mammal { ... }

const simba = new Lion('Симба');
```



```
class Chordate { ... }
class Mammal extends Chordate { ... }
class Carnivore extends Mammal { ... }
class Feline extends Carnivore { ... }
class Lion extends Feline { ... }

const simba = new Lion('Симба');
```

Цепочка может быть любой длины.



super

```
class Lion extends Mammal {
  constructor (name, gender) {
    super(name);
    this.gender = gender;

    super.growHair();
  }
}

new Lion('Симба');
// Симба говорит: у меня растет шерсть
```

MDN

НАСЛЕДОВАНИЕ В ES5

Object.create

```
const foo = {
  bar: 1
};

const baz = Object.create(foo);

baz.__proto__ === foo; // true
baz.bar; // 1
```

Object.create создает новый объект с заданным прототипом.

```
const foo = Object.create(null);

foo.hasOwnProperty; // undefined
```

Часто используется для создания объектов без прототипа.

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

Mammal.prototype.growHair = function () {
  console.log(`${this.name} говорит: у меня растет шерсть`);
};

function Lion (name, gender) {
  this.gender = gender;
}

Lion.prototype.growl = function () {
  console.log(`${this.name} рычит!`);
};
```

Два конструктора, один должен наследоваться от другого.

1. СВЯЗЫВАНИЕ ПРОТОТИПОВ

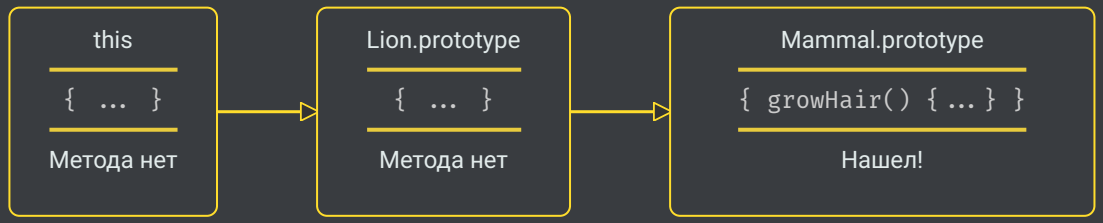
```
function Mammal (name) { ... }
Mammal.prototype.growHair = function () { ... }

function Lion (name, gender) { ... }
Lion.prototype = Object.create(Mammal.prototype);

Lion.prototype.growl = function () { ... };
```

Прототипы связываются в цепочку.

```
function Mammal (name) { ... }
function Lion (name, gender) {
  // Тут прочий код
  this.growHair();
}
Lion.prototype = Object.create(Mammal.prototype);

const simba = new Lion('Симба', 'm');
// Симба говорит: у меня растет шерсть
```

Благодаря этому возможно использовать методы родительского класса в конструкторе-ребенке.

ЦЕПОЧКА ПРОТОТИПОВ



1.5. СВОЙСТВО constructor

```
function Mammal (name) { ... }
Mammal.prototype.growHair = function () { ... }

function Lion (name, gender) { ... }
Lion.prototype = Object.create(Mammal.prototype);
Lion.prototype.constructor = Lion;

Lion.prototype.growl = function () { ... };
```

Подробнее о свойстве constructor.

2. ВЫЗОВ РОДИТЕЛЬСКОГО КОНСТРУКТОРА

```
function Mammal (name) { ... }
Mammal.prototype.growHair = function () { ... }

function Lion (name, gender) {
  Mammal.call(this, name);
  this.gender = gender;
}

Lion.prototype = Object.create(Mammal.prototype);
Lion.prototype.constructor = Lion;
Lion.prototype.growl = function () { ... };
```

СРАВНИМ (НЕ ОЧЕНЬ ТОЧНО)

```
class Mammal {
  constructor (name) {
    this.name = name;
  }

  growHair () {
    console.log(`${this.name} говорит: у меня растет шерсть`);
  }
}

class Lion extends Mammal {
  constructor(name, gender) {
    super(name);
    this.gender = gender;
  }

  growl () {
    console.log(`${this.name} рычит!`);
  }
}
```

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

Mammal.prototype.growHair = function () {
  console.log(`${this.name} говорит: у меня растет шерсть`);
};

function Lion (name, gender) {
  Mammal.call(this, name);
  this.gender = gender;
}

Lion.prototype = Object.create(Mammal.prototype);

Lion.prototype.constructor = Lion;

Lion.prototype.growl = function () {
  console.log(`${this.name} рычит!`);
};
```

```
class Mammal {
  constructor (name) {
    this.name = name;
  }

  growHair () {
    console.log(`${this.name} говорит: у меня растет шерсть`);
  }
}

class Lion extends Mammal {
  growl () {
    console.log(`${this.name} рычит!`);
  }
}
```

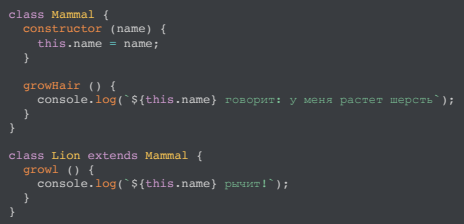
```
function Mammal (name) {
  this.name = name;
}

Mammal.prototype.growHair = function () {
  console.log(`${this.name} говорит: у меня растет шерсть`);
};

function Lion (name) {
  Mammal.call(this, name);
}

Lion.prototype = Object.create(Mammal.prototype);

Lion.prototype.constructor = Lion;

Lion.prototype.growl = function () {
  console.log(`${this.name} рычит!`);
};
```

Конструктор у класса можно опустить, если он не нужен.

for .. in

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

Mammal.prototype.growHair = function () {
  console.log(`${this.name} говорит: у меня растет шерсть`);
};

function Lion (name) {
  Mammal.call(this, name);
}

Lion.prototype = Object.create(Mammal.prototype);

Lion.prototype.constructor = Lion;

Lion.prototype.growl = function () {
  console.log(`${this.name} рычит!`);
};
```

```
const simba = new Lion('Симба');

for (let key in simba) {
  console.log(key);
}

// name
// constructor ?!
// growl ???!!!
// growHair  ????!??!?!?!?!??!
```



```
class Mammal {
  constructor (name) {
    this.name = name;
  }

  growHair () {
    console.log(`${this.name} говорит: у меня растет шерсть`);
  }
}

class Lion extends Mammal {
  growl () {
    console.log(`${this.name} рычит!`);
  }
}
```

```
const simba = new Lion('Симба');

for (let key in simba) {
  console.log(key);
}

// name
```

В классах все методы прототипа автоматически получают флаг `enumerable: false`

СРАВНИМ ЕЩЕ РАЗ (НО ВСЁ ЕЩЕ НЕ ТОЧНО)

```
class Mammal {
  constructor (name) {
    this.name = name;
  }

  growHair () {
    console.log(`${this.name} говорит: у меня растет шерсть`);
  }
}

class Lion extends Mammal {
  constructor(name, gender) {
    super(name);
    this.gender = gender;
  }

  growl () {
    console.log(`${this.name} рычит!`);
  }
}
```

```
function Mammal (name) {
  if (!(this instanceof Mammal)) {
    throw new Error("Mammal needs to be called with the new keyword");
  }

  this.name = name;
}

Object.defineProperties(Mammal.prototype, {
  growHair: {
    enumerable: false,
    value: function () {
      console.log(`${this.name} говорит: у меня растет шерсть`);
    }
  }
})

function Lion (name, gender) {
  if (!(this instanceof Lion)) {
    throw new Error("Lion needs to be called with the new keyword");
  }

  Mammal.call(this, name);
  this.gender = gender;
}

Lion.prototype = Object.create(Mammal.prototype);

Object.defineProperties(Lion.prototype, {
  constructor: {
    enumerable: false,
    value: Lion
  },
  growl: {
    enumerable: false,
    value: function () {
      console.log(`${this.name} рычит!`);
    }
  }
})
```

По всем вопросам пишите в телеграм:

- Общий чат
- @markitosha