ООП в JavaScript

Прототип объекта

— Каждый объект имеет *скрытый* атрибут `[[Prototype]]`

— `[[Prototype]]` может указывать на объект-*прототип* или быть `null`

— **Важно**: `[[Prototype]]` не равен свойству `prototype` объекта!

— Последовательность прототипов называется *цепочкой прототипов*

— `Object.create()` создает новый объект с заданным прототипом

```
// obj - пустой объект, его прототип - это объект { x: 1, y: 2 }
let obj = Object.create({ x: 1, y: 2 });

let obj2 = Object.create(null);             // obj2 не имеет прототипа
let obj3 = Object.create(Object.prototype); // obj3 - обычный объект
```

Чтение и запись свойств

— При чтении свойства JS вначале ищет его в исходном объекте, затем по цепочке прототипов

— Присвоение значения свойству **всегда** выполняется в исходном объекте

```
let obj0 = { a: "a0", b: "b0", c: "c0" },
    obj1 = Object.create(obj0),
    obj2 = Object.create(obj1);

obj1.a = "a1"; obj1.b = "b1";
obj2.a = "a2";

console.log(
  obj0.a, obj0.b, obj0.c); // -> a0 b0 c0
console.log(
  obj1.a, obj1.b, obj1.c); // -> a1 b1 c0
console.log(
  obj2.a, obj2.b, obj2.c); // -> a2 b1 c0
```

```
let a = "a0", b = "b0", c = "c0";

function f1() {
  let a = "a1", b = "b1";

  if (true) {
    let a = "a2";
    console.log(a, b, c); // -> a2 b1 c0
  }
}
```

`this`: вызов метода объекта

— При вызове метода объекта `this` указывает на этот объект

— Если метод находится в прототипе объекта, `this` все равно указывает на изначальный объект

```
let obj = {
    a: 2,
    b: 3,
    add() { return this.a + this.b; }
};
obj.add();      // 5
obj["add"]();   // 5

let mulF = function() {
    return this.a * this.b;
}
obj.mul = mulF;

mulF();    // TypeError, this === undefined
obj.mul(); // 6 все ок, this === obj
```

```
let obj2 = {
    a: 10,
    b: 100,
    div() { return this.b / this.a; }
};
Object.setPrototypeOf(obj, obj2);
obj.div(); // 1.5

// метод div берется из прототипа obj2,
// но this указывает на изначальный объект
```

Установка прототипа

— Есть 3.5 способа задать прототип объекта

— `Object.create(protoObj)`

— `Object.setPrototypeOf(obj, protoObj)` - установить прототип

— Функция-конструктор и оператор `new`

— `obj.__proto__` - свойство-указатель на прототип (*legacy*)

```
let a = {x: 1, y: 2},
    b = {x: 3, y: 4},
    c = Object.create(a);

console.log(c.__proto__ === a);              // true
Object.setPrototypeOf(c, b);
console.log(Object.getPrototypeOf(c) === b); // true
```

Что такое класс?

— *Класс* - шаблон, по которому создаются объекты.

— Содержит код инициализации новых объектов (конструктор)

— Содержит описание поведения (методы)

— Содержит статические методы/переменные (мало чем отличаются от глобальных)

— Определяет *тип данных*

Классы в JS

— В JS до недавних пор не было классов как *синтаксиса*

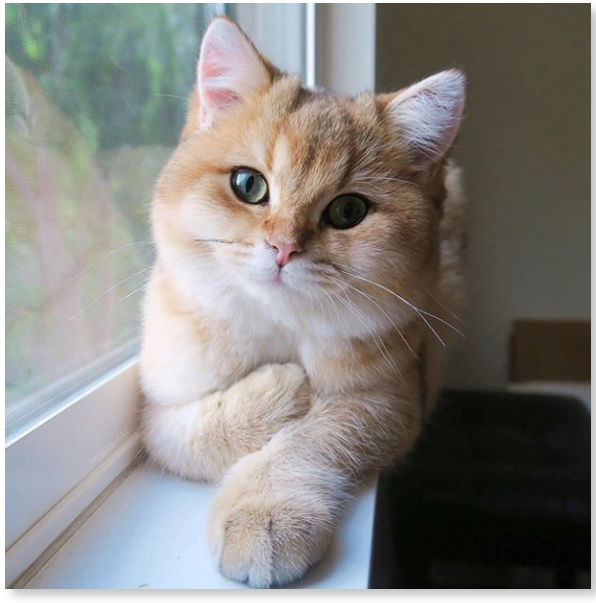
— Однако они всегда присутствовали как *концепция*

— Класс в JS - это множество объектов, имеющих один и тот же прототип

— Собственные данные хранятся в собственных свойствах объекта

— Методы/переменные/константы класса хранятся в прототипе

Котику, де ієрархія класів в прототипному наслідуванні?



Прототипное наследование

```
// Наивная фабрика объектов
function createRange(from, to) {
  return {
    diff() { // Метод объекта
      return this.to - this.from;
    },
    // Данные объекта
    from: from,
    to: to
  };
}

let myRange = createRange(1, 10);
myRange.diff();  // -> 9
```

```
// Выносим методы класса в объект-прототип
let rangePrototype = {
  diff() { return this.to - this.from; }
}
// Инициализация данных в фабрике
function createRange(from, to) {
  let obj = Object.create(rangePrototype)
  obj.from = from;
  obj.to = to;
  return obj;
}

let myRange = createRange(1, 10);
myRange.diff();  // -> 9
```

```
function Range(from, to) {
    this.from = from;
    this.to = to;
}
Range.prototype.diff = function() {
    return this.to - this.from;
};
let myRange = new Range(1, 10);
myRange.diff();  // -> 9
```

Функции-конструкторы и оператор new

— Имя конструктора (класса) принято писать с заглавной буквы

— При вызове с оператором `new`:

Создается новый объект `obj` с прототипом `Constr.prototype`

Выполняется код конструктора с `this` указывающим на новый объект

Если конструктор вернул значение - оно станет результатом вызова

Иначе результатом вызова станет объект `obj`

```
let obj = new Constr(arg1, arg2);

// это то же самое что написать:
let obj = Object.create(Constr.prototype);
let result = Constr.call(obj, arg1, arg2);
obj = result === undefined ? obj : result;
```

Принадлежность к классу

— `[obj] instanceof [Constr]` - проверяет, находится ли свойство `Constr.prototype` в цепочке прототипов `obj`

— `Constr.prototype.constructor` - ссылка на сам конструктор

— `Constr.prototype.isPrototypeOf(obj)`

```
let r = new Range(1, 10),
    r2 = Object.create(Range.prototype);

r instanceof Range;       // -> true
r2 instanceof Range;      // -> true

r.constructor === Range;  // -> true
r2.constructor === Range; // -> true

Range.prototype.isPrototypeOf(r);  // -> true
Range.prototype.isPrototypeOf(r2); // -> true
```

Утиная типизация

— Класс описывает отдельный тип данных

— В JS нельзя указать типы параметров функции

— Функция работает с любым объектом с нужными полями и методами

— Вместо "Что это за объект?" спрашиваем "Что он может делать?"

```
function logDifference(range) {
    console.log(range.diff());
}

let r = new Range(1, 10);
logDifference(r);  // => 9

let r2 = {
    diff() { return 27; }
}
logDifference(r2); // => 27
```

```
let dateRange = new Range(
    new Date(Date.now() - 3600000),
    new Date()
);
dateRange.diff();  // -> 3600000
```

Вызов конструктора без new

— Функцию можно вызвать как конструктор или как обычную функцию

— `new.target` определяет была ли вызвана функция с `new`

— Можно запретить вызов конструктора без `new`, или "перегрузить" его

```
function Range(from, to) {
    if (new.target === undefined) {
        // вызван без new, преобразуем
        // массив из аргумента from в тип Range
        return new Range(from[0], from[1]);
    }
    // вызван с new, обычный конструктор
    this.from = from;
    this.to = to;
}
Range.prototype = { . . . }
let r = new Range (1, 10); // ok
let r2 = Range([1, 15]);   // ok!
r2.diff();                 // => 14
```

Приватные поля и методы

— Объявляются в замыкании конструктора

— Для каждого нового объекта создаются новые дубликаты методов

— Концептуально то же самое что и паттерн "модуль"

— Обычно приватные поля просто называют с underscore: `_from, _to`

```
// переменные from и to - приватные, и доступны только
// в замыканиях функций, объявленных в конструкторе
function Range(from, to) {
    // приватный метод
    const privateIncludes = (x) => from <= x && x <= to;

    // публичные методы
    this.diff = () => to - from;
    this.includes = (x) => privateIncludes(x);
}

Range.prototype = {
    someMethod() {
        // здесь нет доступа к from и to!
    }
};
```

Статические поля и методы

— Принадлежат самому классу, а не объектам класса

— Хранятся в JS как свойства функции-конструктора

```
function Range(from, to) { ... }
Range.prototype = { ... };

Range.MY_CONST = 15;
Range.myStaticMethod = function() { ... };
```

Наследование

— сводится к построению цепочек прототипов

— методы родительских классов доступны через цепочку прототипов

```
function Parent() { ... }
Parent.prototype = { ... }

function Child() { ... }
// нужно чтобы объект Parent.prototype был прототипом Child.prototype

Child.prototype = Object.create(Parent.prototype);
Object.assign(Child.prototype, {
  // собственные методы класса Child
})
```

Наследование

```
// Родительский класс Shape
function Shape(x, y) {
  this.x = x;
  this.y = y;
}
Shape.prototype = {
  getCenter() { return [this.x, this.y]; },
  getName() { return "Shape"; }
};

// Дочерний класс Circle
function Circle(x, y, r) {
  // Вызов родительского конструктора
  Shape.call(this, x, y);
  this.r = r;
}
```

```
Circle.prototype =
                Object.create(Shape.prototype);

Object.assign(Circle.prototype, {
  getArea() {
    return Math.PI * this.r ** 2;
  },
  getName() {
    // вызов родительского метода
    let p = Shape.prototype.getName.call(this);
    return `${p}|Circle`;
  }
});
```

```
let c = new Circle(2, 3, 5);
c.getArea();   // => 78.5398...
c.getCenter(); // => [2, 3]
c.getName();   // => "Shape|Circle"
```

Классы ECMAScript2015

```
class Shape {
  constructor(x, y) {
    // new.target хорошо работает в ES6-стиле
    if (new.target === Shape) {
      throw new Error("Shape is abstract!");
    }
    this.x = x;
    this.y = y;
  }

  getCenter() { return [this.x, this.y]; }
  getName()   { return "Shape"; }
}

typeof Shape; // -> "function"
```

```
class Circle extends Shape {
  constructor(x, y, r) {
    super(x, y); // вызов конструктора Shape
    this.r = r;
  }
  getArea() { return Math.PI * this.r ** 2; }
  getName() {
    // вызов Shape::getName()
    return super.getName() + "|" + "Circle";
  }

  // статический метод
  static createCircle() {
    return new Circle(0, 0, 1);
  }
}
```

```
let obj1 = new Shape(4, 5);    // Error
let obj2 = Circle.createCircle();
console.log(obj2.getCenter()); // [0, 0]
console.log(obj2.getName());   // "Shape|Circle"
```

Классы ECMAScript2015

— "Синтаксический сахар" для прототипного наследования:

— Конструктор нельзя вызвать без `new`

— Можно вызвать родительский метод через `super.methodName()`

— Можно вызвать родительский конструктор через `super()`

— Все методы работают в Strict Mode