ООП в JavaScript

Классы

- В JS до недавних пор не было классов как *синтаксиса*
- Тем не менее, классы всегда присутствовали в языке как *концепция*
- Класс в JS - это множество объектов, имеющих один и тот же прототип
- Собственные данные хранятся как собственные свойства объекта
- Методы/переменные/константы класса хранятся в прототипе

Прототипное наследование

```
// Наивное создание объектов
function createRange(from, to) {
  return {
    from: from,
    to: to,
    diff() { return this.to - this.from; },
    includes(x) {
      return this.from <= x && x <= this.to;
    }
  };
}
let myRange = createRange(1, 10);
myRange.diff();  // => 9
```

```
// Выносим методы класса в отдельный объект
let rangePrototype = {
  diff() { return this.to - this.from; },
  includes(x) {
    return this.from <= x && x <= this.to;
  }
}
function createRange(from, to) {
  let obj = Object.create(rangePrototype);
  obj.from = from;
  obj.to = to;
  return obj;
}
let myRange = createRange(1, 10);
myRange.includes(5);  // => true
```

```
function Range(from, to) {
    this.from = from;
    this.to = to;
}
Range.prototype = {
  diff() { return this.to - this.from; },
  includes(x) { return this.from <= x && x <= this.to; }
};
let myRange = new Range(1, 10);
myRange.includes(11);  // => false
```

Функции-конструкторы и оператор new

- Имя конструктора (класса) принято писать с заглавной буквы
- Вызывается с оператором new (но может и без)
- При вызове с `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;
```

`Object.create()` через функцию-конструктор

- Метод `Object.create()` появился только в ECMAScript5
- До этого создать объект с заданным прототипом можно было только через функцию-конструктор

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

let obj1 = {};
let obj2 = objectCreate(obj1);
obj1.isPrototypeOf(obj2); // true
```

Принадлежность к классу

- `[obj] instanceof [Constr]`- проверяет, находится ли свойство `Constr.prototype`в цепочке прототипов `obj`
  - не проверяет, был ли объект действительно создан конструктором
  - закрепляет использование имени конструктора как имени класса
- `Constr.prototype.constructor`- ссылка на сам конструктор `Constr`
- `Constr.prototype.isPrototypeOf(obj)`

```
let r = new Range(1, 10),
    r2 = new 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

- В статически типизированных языках класс описывает тип данных
- В JS нельзя указать с какими типами или классами работает функция
- Вместо этого функция может работать с любым объектом/типом, имеющим нужные поля/методы:
- Вместо вопроса "Что это за объект?" мы спрашиваем "Что он может делать?"

```
let r = new Range( { valueOf() {return 15;} }, { valueOf() {return 25;} });
r.diff();  // => 10;

let r2 = new Range(new Date(Date.now() - 3600000), new Date());
r2.diff(); // => 3600000;
```

Вызов конструктора без new

- Функция может быть вызвана как конструктор и как обычная функция
- Раньше использовали проверку `this instanceof Constr`
- Сейчас есть более надежный метод: `new.target`
- Можно запретить вызывать конструктор без `new`, или "перегрузить" его

```
function Range(from, to) {
    if (!(this instanceof Range))
        throw new Error('Called without new!');
    this.from = from;
    this.to = to;
}
Range.prototype = { . . . };
let r = new Range (1, 10); // ok
let r2 = Range(1, 10);     // Error

// осадочек остался:
Range.call(r, 5, 10);
r.diff();  // => 5
```

```
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 объекта приходится создавать новые экземпляры методов
- обычно "приватным" полям просто дают имена начиная с underscore: `_from, _to`

```
function Range(from, to) {
    function privateIncludes = (x) => from <= x && x <= to;

    this.diff = function() {
        return to - from;
    };
    this.includes = function(x) {
        return privateIncludes(x);
    };
}
Range.prototype = {
    someMethod() {
        // здесь нет доступа к from и to!
    }
};
```

Статические поля и методы

- Принадлежат 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 в нужном порядке
- методы родительских классов доступны через цепочку прототипов

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

```
function Circle(x, y, r) {
  // Вызов родительского конструктора
  Shape.call(this, x, y);
  this.r = r;
}
// неправильно: Circle.prototype = new Shape();
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'
```

Примеси (mixins) и трейты (traits)

- Набор методов и свойств для добавления в другие объекты
- Для добавления примеси ее добавляют ("подмешивают") в прототип

```
// Mixin содержит только методы
let RoundMixin = {
    area() {
        return Math.PI * this._radius ** 2;
    }
};

// Trait может объявлять собственные данные
let RoundableTrait = {
    radius(r) {
        return r === undefined
            ? this._radius
            : this._radius = r;
    }
};

// Добавляем метод area к трейту
Object.assign(RoundableTrait, RoundMixin);
```

```
function RoundButton(label, radius) {
    this._label = label;
    this._radius = radius;
}
// Подмешиваем RoundMixin в прототип RoundButton
Object.assign(RoundButton.prototype, RoundMixin);
let a = new RoundButton("text", 5);
a.area(); // => 78.5398...

function Cat(name) { this._name = name; }
// Подмешиваем RoundableTrait в прототип Cat
Object.assign(Cat.prototype, RoundableTrait);

let b = new Cat("Coltrane a.k.a. Snowball IV");
// Cat теперь может работать с полем _radius
b.radius(5); // круглокот
b.area();    // 78.5398...
```

Абстрактные классы и методы классов

- Мы можем проверять как создается объект, и предпринять меры
- Или можем ничего не предпринимать :)

```
function ParentClass(param1) {
    // new.target плохо работает с ES5-стилем
    if (this instanceof ParentClass)
      throw new Error('ParentClass is abstract!');

    this.a = param1;
}

ParentClass.prototype = {
    method() {
        throw new Error('method is abstract!');
    }
};
```

```
function ChildClass(param1, param2) {
    ParentClass.call(this, param1);
    this.b = param2;
}

ChildClass.prototype = Object.create(ParentClass)

ChildClass.prototype.method = function() {
    return this.a + this.b;
};
```

```
let obj1 = new ParentClass('a'),          // Error
    obj2 = new ChildClass('a', 'b');      // ok
console.log(obj2.a);          // 'a'
console.log(obj2.b);          // 'b'
console.log(obj2.method());   // 'ab'
ParentClass.prototype.method.call(obj2);  // Error
```

Классы ECMAScript2015

- Простой и более привычный синтаксис объявления классов
- "Синтаксический сахар" для прототипного наследования с множеством удобств:
  - Конструктор нельзя вызвать без `new`
  - Нельзя объявлять свойства-данные (можно только методы)
  - Можно вызввать родительский метод через `super.methodName()`
  - Можно вызвать родительский конструктор через `super()`
  - Все методы не могут вызываться с `new`, неперечислимы и работают в strict режиме
  - Правильное наследование от встроенных типов данных

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'; }
}
```

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